DOCUMENT RESUME

ED 085 254 SE 017 076

AUTHOR	Rawson, Freeman L., III
TITLE	Set-Theoretical Semantics for Elementary Mathematical Language.
INSTITUTION	Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
REPORT NO	TR-220
PUB DATE	7 Nov 73
NOTE	130p.; Psychology and Education Series
EDRS PRICE	MF-$0.65 HC-$6.58
DESCRIPTORS	Computers; *Linguistics; Linguistic Theory; *Mathematical Linguistics; Mathematical Vocabulary; *Mathematics Education; *Programing Languages; Semantics; *Structural Linguistics

ABSTRACT

The development of computer language and analogs capable of interpreting and processing natural language found in elementary mathematics is discussed. Working with linguistic theories in combination with the special characteristics of elementary mathematics, the author has developed algorithms for the computer to accomplish the above task. (JP)

ED 085254

SET-THEORETICAL SEMANTICS FOR ELEMENTARY MATHEMATICAL LANGUAGE

BY

FREEMAN L. RAWSON, III

SCOPE OF INTEREST NOTICE
The ERIC Facility has assigned this document for processing to: SE EM
In our judgement, this document is also of interest to the clearinghouses noted to the right. Indexing should reflect their special points of view.

U S DEPARTMENT OF HEALTH EDUCATION & WELFARE NATIONAL INSTITUTE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

TECHNICAL REPORT NO. 220

NOVEMBER 7, 1973

PSYCHOLOGY AND EDUCATION SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

SE 017 076

FILMED FROM BEST AVAILABLE COPY

ED 085254

SET-THEORETICAL SEMANTICS FOR ELEMENTARY MATHEMATICAL LANGUAGE

by

Freeman L. Rawson, III

TECHNICAL REPORT NO. 220

November 7, 1973

PSYCHOLOGY AND EDUCATION SERIES

Reproduction in Whole or in Part Is Permitted for

Any Purpose of the United States Government

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

STANFORD UNIVERSITY

STANFORD, CALIFORNIA

Table of Contents

Chapter			Page
I.	Introduction		1
	I.1	Outline of the Thesis	1
	I.2	The Fragment of English Handled	2
	I.3	A Theory of Language as a Basis for a Question Answerer	3
	I.4	A Note on the History of the Program	5
	I.5	Prerequisites	9
II.	Linguistic Aspects		10
	II.1	Preliminary Notions About Grammar	10
	II.2	The Basic Semantic Theory	12
	II.3	Constructive Set Theory and Its Role	15
	II.4	The Control Structure View Of Natural Language	20
	II.5	Semantic Transformations and Surface Structure	23
	II.6	The Deep Structure	25
	II.7	A Detailed Transformational Example	29
	II.8	Program Schemata	32
	II.9	The Restrictions Upon Schemata	39

II.10 The Relative Power of the Non-Transformational and Transformational Schemata . . . 44

II.11 A Final Example 46

III. Aspects of the Actual Implementation 50

III.1 Concepts From LISP 50

III.2 Types of Functions to be Implemented 54

III.3 Side Effects 55

III.4 The LISP Calling Sequence 58

III.5 Backtracking 61

III.6 Interrupts and Demons 63

III.7 The Implementation of Data Typing 64

III.8 Mathematical Types 69

III.9 Arithmetical Relations 70

III.10 The Set Theoretical Functions: The I Function . . 72

III.11 The Other Set Theoretical Functions 76

III.12 The Verb 'Have' in an Existential Context 79

III.13 Two Other Constructions Using 'Have' 82

III.14 The Importance of the Run-time Creation of Functions 83

IV. Information Extraction and Heuristics 85

IV.1 Introduction to Information Extraction 85

IV.2 Resolution Is Not a Suitable Basis For Information Extraction 87

IV.3 The Application of Information Extraction To How Questions 90

IV.4 Information Extraction As an Aid to Problem Solving 93

IV.5 Mathematical Information To Be Used By Information Extraction . . . 94

IV.6 The Heuristics Actually Used in the Semantic Evaluator 95

IV.7 Heuristics for the Set Theoretical Functions . . . 96

IV.8 Heuristics For Doing Arithmetic 97

IV.9 The Justification of the Heuristics 100

V. Comparisons With Other Work 103

V.1 Machine Translation 103

V.2 The History of Question-Answering Systems 104

V.3 The Work of Winograd on the BLOCKS Program 105

V.4 The Predicate Calculus as Deep Structure--The Work of Sandewall 108

V.5 The Psychological Model Approach of Schank 109

V.6 The Work of Woods on Natural Language Processing 113

Index 115

References 119

Acknowledgments

This dissertation is largely the result of the patience and hard work of Robert L. Smith, Jr., who supervised all phases of this undertaking, and who provided most of the ideas and much of the programming upon which it is based. He also read and criticized preliminary versions of this thesis.

The grammar that is used by system was written by Nancy W. Smith. She also provided many of the ideas about semantics which were used.

I am grateful to Professor Patrick Suppes for his general supervision of the project, for serving as my thesis advisor, for suggesting many important improvements to this report, and for providing the facilities of the Institute for Mathematical Studies in the Social Sciences for carrying on the work described herein.

My general education about the programming language LISP benefited greatly from conversations with David R. Levine. I am also indebted to Rainer W. Schulz for modifying and maintaining TENEX, and for his patience with my endless questions about the system.

I am grateful to Professors J. M. E. Moravscik and Dov Gabbay for their willingness to serve on my doctoral reading committee and for their patient reading of my efforts at dissertation writing.

This thesis was written using TEC EDIT, a text editing system

developed by Pentti Kanerva. The processing of this document was done using the PUB document compiler, which was written at the Stanford Artificial Intelligence Project by Lawrence Tesler, and which was modified to run at IMSSS by Robert Smith. The entire document was reproduced on an IBM MCST using a program written by Robert Smith.

The research reported in this paper was partially supported by Grant EC-443X4 from the National Science Foundation.

Chapter I

Introduction

This thesis reports some work on computational linguistics and question-answering svstems. As is standard in this area only a small fragment of English is handled, and that only incompletely. The work centers on questions about elementary mathematical language, and in particular on elementary number theory. The work is primarily concerned with the problem of dealing with natural language and only secondarily with mathematical problem-solving. Moreover, the system accepts natural language input, but does not produce natural language output: the problem of output in natural language, which is crucial to any truly interactive system, is not touched upon in the present work.

I.1 Outline of the Thesis

The outline of the thesis is to present the linguistic theory first, and then to comment upon the details of the implementation. Included in that portion of the dissertation will be some ideas about the computer science involved in the project--only some of these have actually been implemented. After this there is a discussion of the heuristics that are called by the semantic evaluation procedure and the reasons that such heuristics are necessary. We conclude with a comparative study of some other recent work in the field.

In the remainder of the introduction we shall try to give the reader a flavor of what the project is about, what its limits are, and why it works (or fails to work) as it does. Hopefully, the basic motivation for this type of work is fairly clear: the computer would be a much more useful servant if it could be addressed more easily. While it is our personal belief that the computer is inherently incapable of understanding in any significant way, it is extremely desirable to test this belief in an actual implementation.

I.2 The Fragment of English Handled

The subject matter that this question-answering system deals with is elementary mathematical language. Although it is difficult to define precisely what elementary mathematics is, a perfectly good informal definition is that it is that part of mathematics that is to be found in the elementary school mathematics curriculum. Thus it includes elementary arithmetic, fractions, decimals, percentages, simple operations on units, simple geometry, elementary set-theory, and the solution of simple word problems. The reason that this is a particularly good subject matter for a question-answering system is that the information that is needed as a data-base is relatively compact and is already well-organized. In an area like geography or medicine one of the great problems is to organize the information that the system is to have available in a reasonable way. The relatively elementary nature of the subject matter is important because we are

primarily concerned with machine comprehension of natural language rather than machine problem-solving: the algorithms that are needed to answer the actual questions once the input has been put into the proper form for the computer's consumption are all very well-known and quite trivial. Thus, the subject matter for the questions does not get in the way of the English. Moreover, the semantics of the English are fairly clear: most input sentences in this fragment have intuitively transparent and universally agreed upon meanings.

I.3 A Theory of Language as a Basis for a Question Answerer

The basis for the processing of the natural language input is provided by a theory of natural language: this point is very important, for many question-answering systems for elementary mathematics have been written, and many of these accept natural language (or something close to natural language) as input, but none of these systems have, to our knowledge, been based upon a systematic theory of language. This is a key feature of our system. Although this theory will be discussed in some detail later, we shall sketch it now. The basic idea is to parse English much the same way as one would parse an ALGOL program. This means that there must be a context-free grammar for the syntax of the language. Such a grammar has been written by Nancy W. Smith, and in fact, deals with rather more of the English than the system as a whole can handle. Associated with each terminal symbol of the language is a denotation, which is formally a

set of some kind. (Note that we are considering functions and numbers to be special kinds of sets.) With each production rule of the grammar there is associated a set-theoretical function that maps the denotations of the right side of the production rule to the denotation of the symbol on the left hand side of the rule. The setup is essentially that used by Irons [1] and Knuth [2] to analyze the semantics of computer languages. Using the syntax tree, the rules associated with each production, and the denotations of the terminal nodes, the denotation of the start symbol can be computed. In the theory as expounded by Knuth, the evaluation is recursive-inside-out like that of the LISP interpreter. The evaluation algorithm that must be used for natural language is considerably more complex than this and will be discussed in detail in Chapter III. Suffice it to say that the system gets most of its linguistic power from this feature.

The limits of the linguistic theory that we are dealing with have not been fully explored, and indeed, its mathematical limitations are not completely known. The present implementation is still in a very incomplete state, and is essentially limited to questions about elementary number theory. Note that while the implementation is biased strongly toward interrogatives, the linguistic theory is a general theory of language. The present project should be regarded less as a question answering system for arithmetic and more as a partial implementation of a natural language system for the computer.

I.4 A Note on the History of the Program

The theory of language that is described above is actually implemented on the PDP10 computer in the form of a rather large program called CONSTRUCT (for constructive set theory, which is the basis for the general view of language that is embodied in the program). Although some parts of the program were written in 1971-2, most of the work that is described in this thesis was done in the first six months of 1973. The syntactic parser and the macro expander that generates the semantic parses with whose evaluation we shall be concerned were written by Robert L. Smith, Jr., and date from the earlier period. The syntactic parser that CONSTRUCT uses is written in a modular fashion so that the grammar upon which its parses are based is read from a disk file and can be written independently of the program itself. There is not only the obvious advantage of modularity to this scheme, but also the equally important feature of assisting in the task of keeping the theory as independent of the machine as possible. As will be apparent throughout this dissertation, we are attempting to develop a theory of language that is implementable on the machine rather than just an implementation. As we mentioned above, the grammar that the program uses was written by Nancy W. Smith: her work was also done during the first six months of 1973. The semantic functions that are attached to the production rules of the grammar are also placed in the disk file with the grammar itself: these functions which are the ones that must be implemented if questions are to be answered were written (as opposed

to being implemented) by Robert Smith and the author although many of them are based upon suggestions made by Nancy Smith: this was also done during early 1973. The actual implementation of these semantic functions was done largely by the author with some help from Robert Smith. Most of the ideas on the theory of language that are used in the program are also due to Robert Smith.

The CONSTRUCT program runs on the PDP10 only under a very special environment--that provided by the TENEX operating system. TENEX is a special operating system developed by Bolt, Beranek, and Newman. Its most notable features are a paged memory allocation scheme, and the ability to run two independent, but intercommunicating processes in the same job with great ease. In TENEX each process is called a fork, and one job may have several forks: there are monitor calls which allow one to map pages from one fork of a job into the memory map of another. Thus, it is possible for the parsing program to run as a controlling process, which reads the natural language input, parses it, forms the semantic parse, and then passes that to the evaluation program which runs as another process in a separate fork. The semantic evaluator, after finishing its computations, passes its output back to the parsing program. At this point in the current implementation what is passed back is simply printed on the output device, but in future versions, we hope to have a program that will produce natural language output to call at this point. The reason that the parsing program and the semantic evaluator must be in different

TENEX forks is that the parsing program is written in SAIL and the semantic evaluator is written in LISP, but the runtime systems of both languages must have a virtual PDP10 to themselves. The use of forks allows us to use together two languages that ordinarily could not be so used. Needless to say, there is also a very great gain in modularity in using two forks, and one of our most important aims has been to make our programs modular and easily extended.

SAIL is a high level extended ALGOL compiler developed for the PDP10 at the Stanford Artificial Intelligence Project by D. Swinehart and R. Sproull. The LISP system that is used was also developed there by John Allen and Lynn Quam. Both of these systems were originally written for a modified Digital 1050 operating system and run under TENEX by means of emulation.

The parsing program compiles the grammar that it obtains from secondary storage, which must be context-free, into a Chomsky normal form grammar, which is actually used to do the parsing. The parsing algorithm is a bottom-up parse using total backup: this means that all possible parses are attempted, and found. There are at least two advantages to this thoroughness in parsing. The first is that the CONSTRUCT program has been used as a portion of a larger system involving speech recognition. A speech recognizer, written by D. Danforth and D. Rogosa, is used to analyze audio input; the results of this analysis are passed to the linguistics program which attempts to parse them and to perform semantic analysis on them. As audio

understanding is so difficult the partial parses must be used in the analysis of the results of the speech recognizer. The other advantage to having partial parses available is that it provides us with a start towards a solution to the habitability problem. The habitability problem is simply the difficulty that arises due to the fact that is not really possible to consider all the English sentences that may be typed at the program in advance and that one cannot write a grammar that handles all grammatical, and also all sensible and quasi-grammatical, inputs. This problem is of course crucial to a fluent program; for further discussion of this almost the only reference is a paper by Coles [3]. During the parse trees are formed representing the derivation of the input string in the grammar: using these trees the semantic parse is formed from the associated semantic functions by means of a macro expansion. This is formatted in such a way as to be acceptable input for the LISP system. As was explained above, the LISP system is in another TENEX fork, and the CONSTRUCT program operates in such a way that LISP reads the semantic parse in precisely the same way that it reads input strings from a teletype. The output from the semantic evaluator is also done by LISP in the usual manner, but is intercepted by the SAIL program in the upper fork before it actually gets to the teletype.

The current LISP implementation is likely to be reprogrammed at some point in the near future in either MLISP2 or L70, which are newer, more powerful derivatives of LISP. The first of these is currently in

use at the Stanford AI Project, and was developed there by Horace Enea and David Canfield Smith. L70 is still being implemented as of this writing by Enea, David Smith, and Lawrence Tesler. The notation that we shall use in the description of the actual implementation in this paper is that of MLISP2 rather than the actual LISP that we used. The language of schemata that is used in the theoretical discussions in the second and third chapters is taken from the dissertation of C. Hewitt [4], and is LISP-like in character, but departs somewhat from both the conventions of LISP and MLISP2.

I.5 Prerequisites

There are a few prerequisites to reading this dissertation. The first is an acquaintance with very elementary ideas from logic such as the notion of an evaluation function and the concept of a model. More significantly, the reader is expected to have a reasonable grasp of the basic elements of the programming language LISP and to be familiar with LISP-like notation.

Chapter II

Linguistic Aspects

As we stated in Section I.3 the basic thrust of this dissertation is to develop and to implement a question-answerer based upon a theory of language. In this chapter we shall discuss our ideas about natural language by introducing the general structure of the theory in some detail (Section II.1 and Section II.2). During the course of this the problem of the nature of the denoted objects that our system must manipulate arises: this is dealt with in Section II.3. After that the key idea of control structures is introduced (Section II.4), and is related to semantic transformations in Section II.5 and Section II.6. Finally, the ideas of schematology are presented in Section II.8, and are put forward as the proper mathematical analysis of our linguistic theory.

It should be borne in mind here that much of what is said in this chapter has been heavily influenced by our actual experience in the implementation of our system. Moreover, the state of the mathematical analysis is still rather incomplete, for the idea of using schemata to analyze linguistic theory is still quite new.

II.1 Preliminary Notions About Grammar

Let V be a finite set of symbols. Then V* is the set of all

finite strings of symbols chosen out of V. Also V+ is V* -{e} where e is the empty string. We shall now define the notions of generative and context-free grammar.

Definition: Let G = <V, T, S, P> be an ordered quadruple. Then G is a generative grammar if V is a finite set (the vocabulary), T is a subset of V (the terminal vocabulary), S is a distinguished element of V that is not in T (the start symbol), and P is a finite subset of V* x V* (the set of productions of G).

The set V-T is called the set of non-terminal symbols, and is often denoted by N. L(G) is the set of all members of T* that can be obtained from S by a sequence of applications of productions in P : we say that a production p is applied to a string t just in case the left-hand side of p matches some substring of t and that substring is then replaced by the right-hand side of p. L(G) is called the language generated by G. A context-free grammar is a particular type of generative grammar--all of its production rules have a single non-terminal on the left-hand side of the production, and a member of V* on the right. For more details about context-free grammars and languages, see [5].

The notion of a context-free grammar was introduced by Chomsky in the mid-1950's to explain some accounts of the syntax of English that he thought were deficient in various ways. Although the notion is used in this thesis to implement a natural language input system for the computer, most of the ideas that are involved parallel ideas that have been developed by compiler writers in the last fifteen years.

II.2 The Basic Semantic Theory

Next we shall introduce the notion of an evaluation: this is derived from the classical model theoretical notion of an assignment to the non-logical symbols, and indeed, serves the same purpose in our semantical theory. We shall assume that a domain for the evaluation is given, and is denoted by D even though the precise nature of this domain is still an open question. It should also be noted that this notion of an evaluation is quite different from the notion of evaluation in LISP, and the two should not be confused. The notion that we are about to define is an abstract model theoretic idea that is relevant only to the semantic theory while the LISP EVAL function is the basic routine that is used by the LISP interpreter to do its computations. (Most of the definitions of this section are taken from [7] or [6].)

Definition: Let D be a non-empty set, let G be a phrase-structure grammar, and let v be a total function from t the terminal vocabulary of G to E, where E is also a non-empty set. Then v is called a valuation function.

While one can certainly use many arbitrary valuation functions, throughout the remainder of this thesis we shall be concerned only with one--the valuation function which gives rise to the intended classical interpretation of elementary mathematics. We shall now introduce some terminology that will prove to be convenient later.

Definition: Let D be a non-empty set, let G be a phrase-

structure grammar, and let v be an evaluation function. Then the ordered pair <D, v> is a model structure for G.

Now we shall develop first intuitively and then more formally the semantical ideas that are involved in our program. With each terminal symbol of a context-free grammar we may associate a denotation: we shall later spell out precisely what kinds of objects these denotations are, but for the time being we trust that the reader will treat them as some kind of primitive. Also with each production of the grammar we shall associate a function from cross-products of denotations to denotations: this is not a new idea as it first appears in the work of Irons in the early 1960's. Each of these functions maps the denotations of the symbols on the right hand side of a production to the denotation for the single symbol on the left hand side of the rule. Note that the requirement that the grammar be context-free is essential to this, for in a context- sensitive grammar there may be more than one symbol on the left hand side of some production rule. In this way the denotations of the nonterminal nodes appearing in a derivation of a string in the language of the grammar may be computed. The denotation of the start symbol is taken to be the denotation of the sentence.

We shall next introduce the notion of a potentially denoting grammar and indicate how this idea can be used to define the concept of the denotation of a non-terminal node of a derivation tree.

Definition: Let G = <V, T, S, P> be a context-free grammar.

Let SEMFUN be a function that assigns to each p in P a set-theoretical function SEMFUN(p). We require that SEMFUN (p) have exactly as many arguments as there symbols on the right-hand side of p. Then G' = <V, T, S, P, SEMFUN> is called a potentially denoting context-free grammar.

In [7] this definition is given in a somewhat more general form, but the above suffices for our purposes. Although we have introduced the notions of set-theoretical function and denotation we have really explain what they are: that task is the function of the next definition.

Definition: Let D be a nonempty set. Then H'(D) is the smallest family of sets such that

(i) D is in H'(D),
(ii) if A and B are in H'(D) then A union B is in H'(D),
(iii) if A is in H'(D) then PA is in H'(D),
(iv) if A is in H'(D) and B is a subset of A, then B is in H'(D).

We define H(D) = H'(D) union {T, F}, with T not equal to F, T and F not in H'(D).

So far we have defined the semantic interpretations of only terminals, but it is our intention that non-terminals should also have denotations or semantic values. These values are defined in terms of the idea of a potentially denoting grammar. It is intended that the denotations of the non-terminals should be members of the hierarchy H(D) defined above.

Definition: Let M be a model structure for a potentially denoting grammar G. The value v of a node N of a derivation tree T in the grammar G with model structure <D, vm> is defined to be:

(i) vm(N) if N is a terminal node
(ii) if N1,...,Nk are the immediate successors of N and the rule P used to derive N1,...,Nk from N has associated with it the function F then the value of the node N is v(N) = F (V(N1),...,V(Nk)).

II.3 Constructive Set Theory and Its Role

We have assumed that the terminal symbols of the grammar are assigned denotations from among the elements of H(D). This means that everything is to be thought of as a set in some sense. Note that neither the machine nor the human can represent every set as an explicit list, for lists such as that of the even numbers are infinite. Such sets must be represented as a characteristic function. Although the representation for a set is completely transparent mathematically, it is quite important in our semantics system. One of the major tasks of the semantic evaluator is to convert from one representation of a set to the other. Usually, our system attempts to convert everything to lists from the characteristic function representation: this is done for reasons of mathematical fluency. In general, it seems to be better to represent a set as an explicit list whenever possible. Mathematically, one gets answers that are intuitively more appealing and somehow more informative. For example, it is possible to determine whether all of the members of the set of even primes are factors of 4 once one has represented this set as the list containing only the number 2. The puzzling thing about this phenomenon is the lack of any easily stated reason as to why one should almost always prefer the list

representation to the characteristic function form. As it turns out, one cannot always list even finite sets in practice due to the constraints of time and memory space; however, similar constraints are also in force for the human so that in fact, most of the sets that actually occur are either relatively small finite sets which are easily listable or are infinite sets that are easily describable.

It is also worth noting that while the conversions between representations for sets seems to be essential to the question answering performance of the system, it is also true that these conversions depend crucially upon the semantics. The reason for this is that the conversions can be made only upon a knowledge of the mathematical properties of the sets being represented: there is no uniform way to determine if an arbitrary recursive set is finite, and hence, can be represented as an explicit list. This fact is a corollary of the unsolvability of the halting problem. If there were a uniform method of determining if an arbitrary recursive set is finite or not, then one could determine this for the set of tape states for a computation of a Turing machine on any input. But then one could determine recursively whether the Turing machine halts on that input, which is impossible. Hence, the semantic evaluator must use its knowledge of what set is involved and what its mathematical properties are in order to determine when it can and should convert from one representation of a set to another.

At this point it should be fairly clear that we have a very

nonstandard set theory in mind as the basis for our semantic theory. While we do not have as yet a complete understanding of this, there are some ideas that we shall use in the sequel that must be explained. It should be noted that our views on set theory are not motivated by abstract philosophical considerations, but rather by the practical necessities of implementing our theory We shall not discuss the abstract philosophy of mathematics involved in this, but rather some of the more concrete aspects that are necessary to our implementation.

In the standard classical set-theory that forms the basis of modern mathematics, sets are very abstract objects: in Godel-Bernays set-theory any class that belongs to another is a set. In general there need be no way to determine algorithmically whether an object belongs to a particular set. In fact it is the case that there are sets of integers that can be specified quite simply that are not recursive, i.e., whose membership problem is not algorithmically decidable. For example, by the unsolvability of the halting problem for Turing machines the set of indices of recursive functions that halt when applied to that index is not recursive. But it is fairly clear that most of elementary mathematics does not involve such esoteric sets of integers; for most of the sets of integers that are considered in elementary mathematics there is a well-known and usually very simple algorithm for deciding the membership problem for that set. For example, it is easily decidable whether a given integer is even or not by dividing it by two and testing for a zero remainder. Indeed, most

of the sets such as the prime numbers, the even numbers, the odd numbers, and so on are defined in terms of such an algorithm for determining if a particular number belongs to the set.

Although our semantics for natural language is based on set-theory, we demand that only recursive or algorithmically defined or constructive sets be used. Not only must we know that there is an algorithm for the membership problem, but also we must have that algorithm in hand: knowledge that such an algorithm exists is not sufficient. Of course in the domain that our actual implementation deals with this is no problem.

This idea of constructive set theory is not the one that seems to be envisioned by the constructivistic philosophers of mathematics, for their idea is based upon philsophical considerations about what a set "really" is, and leads them into the development of a very different mathematics. Our constructivism is much more straightforward and is based upon the necessity of dealing with the computer in a reasonable way: the effects of this upon the way in which our system performs mathematically are essentially non-existent. The question answering system does classical arithmetic, and makes all the standard assumptions that classical mathematicians make about numbers and sets. It does not, for example, incorporate an intuitionistic arithmetic. Rather the constructivist position that we are taking deals not with the subject matter of the question answerer, but rather with the semantical machinery. What we are claiming is that the only kinds of

sets that may appear in the semantic parses that our system produces are sets that are algorithmically defined, and for which we have the algorithms at hand. This is so simply because these are the only sets that we have any hope of representing in our data structures. It should be pointed out also that the fact that we cannot represent non-recursive sets in our data structures seems to be inherent in the nature of the problem rather than any indication of a flaw in the data structures themselves.

Next we shall show that in a certain sense the set-theoretical hierarchy that we have described in the previous section is too strong. The following completely trivial theorem shows that there is a set that our system cannot deal with and that indeed the machine cannot represent in our framework.

Theorem: Let H(D) be as in Section II.2. If D is taken to be the set of all natural numbers, then there is a non-recursive set in H(D).

Proof: Simply notice that since the set of all natural numbers is in H(D), so that all subsets of natural numbers also belong to H(D). Now simply note that there are only countably many recursive sets, but there are uncountably many subsets of the natural numbers.

At this point one might suggest that the definition be modified in some way to eliminate the non-recursive sets. While this can probably be done, the obvious modification fails to provide us with the proper structure. The obvious modification is of course to require

that all of the sets in H(D) be recursive. The difficulty is with the third clause. Clearly, something like power set is needed to handle the escalation of type that may occur in dealing with elementary mathematics. However, the following manifestation of the halting problem prevents us from using the natural analog to clause (iii) of the previous definition.

Theorem: Let A be a recursive set, and let B be the set of all recursive subsets of A. In general B is not recursive.

The proof of this result can be found in Chapter 14 of [5]. This gives a completely deterministic description of the method of computing the values of the nodes of the parse tree and hence of answering questions in our system. However, this is really quite misleading, for the computation process is very much more complex than this description indicates. We shall use the remainder of this chapter to show why it must be so complicated and the next two chapters to indicate how this can be implemented.

II.4 The Control Structure View Of Natural Language

The above definitions can well lead one to believe that things that are parsed together by the rules of the grammar are to be associated as semantic units. This is of course not the case and if it were to be implemented the system would not be able to deal very well with any reasonable fragment of English. To indicate what must actually be done it is necessary to discuss our view of the structure of this fragment of English and to make a number of distinctions.

As we have said before, the semantic objects that we are using are all sets of one kind or another. However, for heuristic purposes we may also regard this fragment of English as consisting of functions, their arguments, and constructions which relate functions to their arguments. There are a number of ways besides application in which functions may be related to arguments, and the English surface construction of a question in this domain can be used in a number of ways to pass arguments to functions. As we shall see later, the verb "have" causes a very different sequence of computational results from the verb "is". There are two things to be emphasized in this analysis: the first is that this is a method of viewing the surface structure of the sentence. While there may be some other reasons for believing that this analysis is either useful or truthful, all that it will be used for here is a metaphor to help explain the way in which our program analyzes its input. The other point that must be made is that the functions that we are discussing are of two quite distinct types, and correspond to two different kinds of functions that we must program in our semantic evaluator.

Before indicating what these functional types are, let us look at a sample.

What are the even multiples of 12? (A)

In the above sentence the words "even" and "multiples" correspond to functions of elementary mathematics which have well-defined algorithms for computation. The number 2 is used as an

argument to the function "multiples", and the result of that computation is passed as an argument to the function "even". The rest of the sentence is used to indicate that some special computation is required to relate properly the output of the application of multiples to 2 to the even function. The reason for this special processing should be obvious--the even function normally tests its argument for membership in a set of numbers, and hence, expects a number as an argument. But the result of the application of the multiple function to 2 is a set of numbers.

The two types of functions that appear in the surface structure of an input sentence are the mathematical functions such as even, factor and odd, all of which have well defined mathematical properties associated with them, and all of which represent operations of elementary mathematics or sets of numbers that are commonly used in elementary mathematics. The other type of function is the mechanism that is used to control the way functions are passed their arguments and the order in which they are called. Since these components of a sentence are also functions, they may be manipulated in exactly the same ways as the other functions in the sentence. One may think of the sentence as having a control structure. This control structure is expressed in the form of a LISP s-expression that is produced by the parsing program and which is passed to the semantic evaluator.

The rest of the discussion that we will present on the linguistic aspects of our system is probably best regarded as a

discussion of the types of control structures that are suitable for the analysis of natural language. Although we have not as yet really completed the study of the mathematics that is needed to describe these control structures, we shall present the beginnings of it as well as some of the important linguistic intuitions behind our work. Despite the fact that much of our work is to be formulated in terms of examples rather than in terms of mathematical theory and despite the fact that the detailed results that we would like to have are not completely worked out as yet, we feel that it is important to present the basic ideas used in our semantic evaluator as well as some analysis of them.

II.5 Semantic Transformations and Surface Structure

There are a number of issues that we wish to cover in our survey of linguistic theory in the rather harsh light of a system for computational linguistics. The first of these is the now overburdened problem of deep structure: the avowed purpose of our work in this area is to make a case for the idea that there is a semantic deep structure. The basic arrangement of our system is that a context-free grammar which is designed in such a way as to lay out the computational structures involved is used to parse the input sentence. By the phrase laying out the computational structures, we mean simply that the grammar is rather detailed in its analysis of English, giving detailed classifications of the syntactic types involved in the sentence. In general structures which lead to very different computational processes should not be parsed in the same way.

For example, the verb "is" leads to a fairly simple and straightforward application of functions to arguments. In a sentence like

Is 2 a factor of 3?

the factor function is applied to 3 to produce a list of factors of 3 and then it is determined whether the list consisting of 2 is a sublist of the list of factors of 3 or not. However, the use of the verb "have" leads to a very different computational process For example, if we take the sentence

Does 2 have a factor of 3?

then we do not apply the factor function to 3, but rather to 2. Then the semantic evaluator checks to see if 3 is among the factors of 2. Although in practice the actual performance of the system is somewhat different from this, the idea sketched above is intuitively correct, and illustrates our point about the grammar The grammar must parse these two questions differently, so that they are seen as having different semantics by the semantic evaluator: this is what we mean when we talk about the idea of the syntax laying out the computational structure. It is clear, for example, that a grammar that used the blanket category VP would not do. (For a further discussion of this issue, see [8]).

Using the productions that were involved in the parse of the input, a semantic representation of the input is produced. We claim that this corresponds to the surface structure semantics. The idea is that the functions associated with the production rules are closely

associated with the grammar, and that the semantic parses that are formed by the parser represent the semantic structure of the sentence prior to the application of semantic transformations.

II.6 The Deep Structure

During the course of the evaluation process certain of the functions that are called have the effect of causing a transformation in the semantic structure. While our implementation never actually transforms the semantic parse into a new one to be evaluated in a more straightforward manner, it could do so, and for the purposes of this discussion is convenient to assume that it does do so. The major characteristic of the transformed semantic parse is simply that things that are parsed together grammatically form semantic units and can be evaluated together.

To clarify this a little, we present a simple example.

Is 2 or 3 even? (B)

It should be intuitively obvious that this sentence is to mean the following:

Is 2 even or is 3 even? (C)

Indeed, if one were to translate (B) into logical symbols as one often does in elementary logic courses, then one might well write down (C) as an intermediate step. It should be clear that (B) and (C) are logically equivalent sentences of English. However, the grammar is such that the most natural syntactic parse lumps the entire phrase "2

or 3" together as a noun phrase, and indeed, this type of example is often used to motivate the introduction of syntactic transformations. However, in our system this problem is handled in the semantics by the use of a special function which produces a transformation of the semantic parse of the sentence. For this particular example, the system in the process of evaluating (B) actually evaluates (C). The details of what these transformations are and how they really work are in the next chapter.

The question immediately arises as to why the transformations are placed in the semantics rather than in the syntax--which is where the linguists have always put them. The first reason why it is necessary to put the transformational component into the semantics is that there is no known way to parse a transformational grammar with any reasonable speed. Although Petrick in [9] did develop an algorithm for such parsing as long ago as 1965, Woods points out in [10] that the time required for computations using the Petrick algorithm is prohibitive. Even though Woods' own augmented transition networks give the power of a transformational grammar and still allow parsing in time close to that needed by Early's algorithm [11] for parsing context-free languages, the Woods method is very different from the methods used by linguists, and is rather less perspicuous than the standard context-free language. The other problem with the Woods parsing scheme is that the parser must be written as a set of procedures in LISP, and therefore, is rather difficult to modify or extend. Thus on purely

practical grounds it was necessary to avoid the use of syntactic transformations.

Yet there are other more compelling reasons for doing without syntactic transformations. The most important of these is simply that it is easier to see what kinds of transformations are needed if one uses semantic rather than syntactic transformations. In the domain that our system is written for, almost all educated adults can agree on the answers to the questions, and indeed, can answer them quite easily. Given these intuitions about how one wants to answer a particular question, it is far easier to write the transformations on the semantics that one wants than to write some syntactic transformations and hope that they do the proper thing semantically. This approach is also simpler in the sense that it is more direct: one just writes the transformational functions that one needs rather than hoping that the proper transformations will be a side-effect of some syntactic transformations.

Given that we accept the idea of semantic transformations, we may well ask what their general form is and what the general form of the semantic deep structure that we imagine to exist is. The general purpose of a semantic transformation is to move functions and their arguments into structural proximity within the semantic deep structure. That is, semantic transformations such as those that are associated with the verb "have" serve to move functions closer to their arguments: this should be apparent from the earlier discussion of this verb. To

put this somewhat differently, a semantic transformation serves to alter the semantic parse tree in such a way as to make the values of nodes "depend" only on the values of their immediate descendents. Thus the general form of a semantic transformation is that of a recursive function that carries semantic parses to semantic parses.

From the above it should now be fairly obvious what the semantic deep structure tree should look like. In an intuitive sense it should be a tree such that one can evaluate its nodes from bottom to top in such a way that the value of any one node is completely determined by the values of its immediate successors in the tree. While this can always be done by means of coding devices, the semantic deep structure should not use such, so that the denotations of all of the nodes of the deep structure tree should have denotations which are relatively simple computationally.

It should be noted in passing that the format of the deep structure tree is not to be radically different from that of the surface structure tree as some have suggested (e.g., Sandewall in [12]). (For further discussion of this point, see Section IV.2 and Section V.4). That is, the deep structure tree is taken to represent a transformation of the surface structure rather than a translation into some formal language such as first order logic. This is important, for while it is possible to see how to do transformations, it is difficult to find reasonable methods for translating sentences of English to, say, first order logic. Although

some heuristic methods are usually presented in introductory logic courses, they do not have sufficient generality or precision to be used as the basis for a question answering system or as the basis for a semantics of natural language. In general, it seems to be the case that the translation problem is harder than the problem of a proper semantics of natural language, and indeed, only once the semantics of natural language are well-understood, will the translation problem then be approachable.

II.7 A Detailed Transformational Example

As much of the preceding has been fairly abstract, we shall now present a rather detailed example of how all of this is to work. We shall use sentence (B) above. The idea is not to give an explicit account of how our current implementation works, as this is discussed in the next chapter, but rather to indicate the behavior of the mechanism of semantic transformations in a particular instance. The parse that is produced by the CONSTRUCT program is :

(S (CHL (LST 2) (LST 3)) (STS EVEN)). (D)

In order to get a good idea of how our transformations do work, we shall trace through a computation with this parse, indicating the important transformational aspects involved.

The evaluation process is a recursive procedure, and is best explained in terms of a stack ST. We shall not really worry about the details of the stack, but rather assume that there are appropriate PUSH

and POP operations: the PUSH operation puts an item on the top of the stack while the POP instruction removes the top item from ST. The input to the computation process is the entire form (D). On input the evaluator tests to determine if (D) is an atomic form, i.e., if (D) is a LISP atom (an identifier). This test fails, so that the first element of the list is pushed onto the stack.

ST = [S]. scanning (CHL (LST 2) (LST 3)).

We shall use diagrams of the above type to keep track of the current state of the semantic evaluator.

The semantic processor now attempts to evaluate the CHL, but again the form is not atomic, and so the rest of the list that was being scanned above must first be evaluated. Since this is the case, CHL is pushed onto ST, so that we have the following computational state.

ST = [CHL S] scanning (LST 2)

Once more the expression being scanned is too complex to be evaluated immediately, so LST is pushed onto ST, and the system is left scanning 2. However, 2 can be immediately evaluated to 2. At this point all of the arguments to LST, which is on the stack, have been evaluated, so that LST may be popped from the stack, and applied to 2. The result of this is simply the form (LST 2). A similar process evaluates (LST 3), so that all of the arguments to CHL have been evaluated, and CHL may be popped from the stack and applied to (LST 2) and (LST 3). Diagrammatically, we have:

```
ST = [S] scanning (STS EVEN)
applying CHL to arg1 = (LST 2) arg2 = (LST 3).
```

At this point the transformational mechanism comes into play: prior to this the evaluation process has been precisely that used by the LISP interpreter. The application of CHL to its arguments has the effect of popping the stack once and writing on the input device the forms:

```
(S (LST 2) (STS EVEN)) (D1)
(S (LST 3) (STS EVEN)) (D2).
```

Then the CHL itself is pushed onto SI. Now following a process that is essentially the same as that above, (D1) and (D2) are evaluated. The following diagram shows this process in some detail.

```
                    ST = [S] scanning (LST 2)
                     ST = [LST S] scanning 2
                         evaluate 2 to 2
                   POP LST and apply to arg1 = 2
                    evaluate (LST 2) to (LST 2)
                   ST = [S] scanning (STS EVEN)
                    ST = [STS S] scanning EVEN
                       evaluate EVEN to EVEN
                 POP STS and apply to arg1 = EVEN
                evaluate (STS EVEN) to (STS EVEN)
          POP S and apply to arg1 = (LST 2) arg2 = (STS EVEN)
       evaluate (S (LST 2) (STS EVEN)) to (TV T)

                    ST = [S] scanning (LST 3)
                     ST = [LST S] scanning 3
                         evaluate 3 to 3
                   POP LST and apply to arg1 = 3
                    evaluate (LST 3) to (LST 3)
                   ST = [S] scanning (STS EVEN)
                    ST = [STS S] scanning EVEN
                       evaluate EVEN to EVEN
                 POP STS and apply to arg1 = EVEN
                evaluate (STS EVEN) to (STS EVEN)
          POP S and apply to arg1 = (LST 3) arg2 = (STS EVEN)
       evaluate (S (LST 3) (STS EVEN)) to (TV NIL)
```

The result of evaluating form (D1) is (TV T) while the outcome of evaluating form (D2) is (TV NIL). Popping CHL from ST, we have as a final result (CHL (TV T) (TV NIL)).

II.8 Program Schemata

We shall next make some attempts to deal with the problem of characterizing the exact mathematical nature of the semantic deep structure, and the precise way in which the transformational component that we envision should work. It should be pointed out that this presentation is only a sketch of the outlines of what we feel can be developed into a theory of semantics.

The first step that must be taken in the analysis of the semantic deep structure is one of abstraction. The mathematical functions are themselves capable of great complexity, but we do not want to claim that simple computations calling relatively complex functions are transformational: rather the additional complexity comes from the way in which arguments are passed to functions and the way in which control flows from one function call to the next. One should note that that this is analogous to the description given earlier of the structure of English in this fragment being the method of passing arguments to functions. The basic idea of the abstraction is replace those functions which are known and which do not have any transformational import by variables. In what follows we should use two types of variables--one for the mathematical functions and one for

the semantic functions that do not have any transformational import, but our theory is not as yet so sophisticated that this needs to be done. This is strictly analogous to the use of program schemata in the mathematical theory of computation. As is well-known, most programming languages are universal in the sense that a program for any recursive function can be written in them. Yet it is clear that some programming languages have features that make them more powerful than others. For example, it is intuitively obvious that ALGOL is more powerful than machine language, but both will allow one to program any recursive function. For this reason program schemata have been developed. By replacing the basic functions by variables an studying only the control structures involved, one can compare the power of programming language features: for an example of this see [13].

It should be pointed out before we actually introduce the schemata themselves that they are presented as an analysis of the functions that are actually called by the semantic evaluator: they are not templates from which these functions were written. Thus, there is the question as to whether the schemata that we write down actually do represent the functions that we claim they represent. This is an assertion that is not subject to any formal checking at the present time: we believe that our analysis is fairly self-evident.

We shall introduce two classes of schemata for use in this endeavor. The first of these is to correspond to the full transformational structure that is used by our system. It should be

noted that this is probably stronger than our semantic evaluator, i.e., there are programs that can be represented in this class of schemata that are not used in, and indeed, should not be used in the programming of a semantic evaluator such as the one that we are discussing. This is analogous to the problem that has plagued linguists for some time--that of finding natural constraints on their transformations. The class of transformational schemata is exactly the class of program schemata that is defined in Hewitt [4] and the following BNF definition of the class is taken directly from that source.

Definition: The BNF syntax for a transformational schema is:

```
                    <program> ::= <term> |
                     <term> ::= <block> |
                                <repeat> |
                                 <again> |
                                 <exit> |
          (if <term> then <terms> else <terms>) |
                               <assignment> |
                                  false |
                             <literal-string> |
                               <identifier> |
                               <function-call>
                  <block> ::= (block <body>)
        <assignment> ::= (<identifier> "<-" <term>)
                 <repeat> ::= (repeat <body>)
<function-call> ::= (<uninterpreted-function> <arguments>) |
                    (is <term> <term>) |
  (call (<uninterpreted-function> <arguments>) <function>)
           <again> ::= (again) | (again <name>)
     <exit> ::= (exit <name> <terms>) | (return <terms>)
        <body> ::= <name> <declaration> <terms> |
                  <declaration> <terms>
           <terms> ::= <term> | <term> <terms>
            <declaration> ::= (<identifiers>)
        <arguments> ::= <empty string> | <terms>
<identifiers> ::= <empty string> | <identifier> <identifiers>
    <identifier> ::= <letter> | <letter> <alphanumeric>
          <alphanumeric> ::= <letter> | <digit>
```

A number of comments need to be made about this definition--mostly in the form of giving the semantics of some of the more unusual constructions, and describing some of the terminology that is common in the field, but rather non-standard outside of it.

The most striking terminological problem is with what we shall call variables. The variables of a schema are the identifiers that appear in it: they serve as memory locations that are used by the schema in the course of a computation. Usually, the variables that appear in a schema are divided into two classes--the input variables and the working variables. The input variables are those that appear following the name of the schema on the left hand side of the defining equality sign: the working variables are all of the rest. Sometimes the term registers is used instead of the term variables.

The evaluation of these schemata is basically a very straightforward process. The execution begins with the first statement, and proceeds sequentially, one statement at a time, with the usual exception that there are statements that when executed affect the order of evaluation. It should be noted that for the class of non-transformational schemata that will be introduced shortly there are no legal statements that change the flow of control, so that each statement in such a schema may be executed precisely once.

These schemata are not recursive, so that a function may not call itself: there are recursive schemata, but they are even more powerful than transformational schema as there exist recursive schemata

which cannot be programmed using only the features of transformational schemata. (There is a proof of this result in Hewitt [4].) These recursive schemata will not, therefore, be considered here. The basic idea is that a transformational schema represents an ordinary iterative program with conditionals and assignment statements. The looping power of these schemata is supplied by the repeat feature. First any statement that is associated with the repeat itself is executed precisely once: usually this is an assignment statement that initializes some variable. Then the body of the repeat is executed until a return statement is encountered; when a return is executed the program returns to the smallest containing block with the indicated values. Thus, the action of the return statement is strictly analogous to that of the RETURN function in LISP. The blocks of program schema may be named and the exit statement allows one to return from the named block with appropriate values: this generalization of the return is somewhat like the done construct in SAIL. Also it should be pointed out that the predicate "is" is used to test for equality to a finite number of distinguished constants. Otherwise, the equality relation is not available as an interpreted feature of transformational schemata. The uninterpreted functions may be called in the usual fashion: first their arguments are evaluated left to right and then the function name is applied to the arguments to produce a value: a function letter accepts only a fixed number of arguments. This last feature is, of course, at variance with the practice in LISP, but it makes the

mathematics of the situation easier. Finally, it should be noted that the LISP convention with regard to truth values is used, i.e., there is some object in the domain of the computation that is to serve as false, and everything else is considered true. This device allows us to dispense with the usual distinction between predicates and functions. All of the other constructs should be self-explanatory.

In order to make this a little more concrete, we shall give a rather simple example of a program schema.

```
(g x) = (repeat ((y <- x))
(if (is x "nil") then (return y))
        (y <- (f x))
        (x <- (h x))).
```

This schema initializes y to the input value of x, and then enters a loop. If x is equal to "nil", then the value of y is returned as the value of the schema. Otherwise, the value of y is set to the value of (f x), and that of x is set to (h x), and the loop is started again.

Now we are ready to introduce the notion of a non-transformational schema. Such schemata are all transformational schemata of a very restricted type, so that all of the restrictions that applied to transformational schemata apply to non-transformational schemata as well. In particular, these schemata are non-recursive and use only uninterpreted functions of a fixed number of arguments. The basic idea of a non-transformational schema is that it is a sequence of assignment statements, followed by a return statement that indicates the value of the computation as a whole. This class of schemata is too weak to be of interest to specialists in the mathematical theory of

computation, and consequently is not to be found in the standard papers on the subject. The interest of these schemata is simply that they seem to be a reasonable formalization of the notion of a non-transformational semantic evaluation.

Definition: The BNF definition for the syntax of the class of non-transformational schemata is as follows:

```
<program> ::= (block <terms> <return>) | <term>
<terms> ::= <term> <terms>
<term> ::= <assignment> | <function call>
<assignment> ::= (<identifier> "<-" <term>)
<function call> ::= (<uninterpreted function> <arguments>)
<arguments> ::= <argument> | <argument> <arguments> |
                <empty>
<argument> ::= <identifier>
<return> ::= (return <identifier>).
```

It should be clear from this definition that the non-transformational schemata form a subclass of the class of transformational schemata.

A simple example should serve to make this much more concrete. This particular program schema assigns to y the value of f applied to x, and then assigns that to z, which is returned as the value of the whole schema.

Example of Non-transformational Schema

```
(g x) = (block
  (y <- (f x))
    (z <- y)
  (return z)).
```

We shall briefly consider how this particular schema is to be evaluated in order to clarify the evaluation process for such schemata. First the assignment statement assigning to the variable y the value (f x) is executed, and then this value is assigned to z. The schema then returns z, and since there are no more statements to execute, halts.

II.9 The Restrictions Upon Schemata

Before showing the relationship between the class of transformational schemata and the sub-class of non-transformational schema, we shall first consider a bit more fully the reasons for the restrictions that are placed upon both classes. These restrictions are quite important both formally and linguistically. Our analysis of the computational processes involved in the semantics of natural language is such that the forms that are barred by the restrictions on transformational schemata do not occur in the actual semantic computations.

There are two important restrictions that we wish to discuss: the first of these applies to both transformational and non-transformational schemata. Both classes are restricted to non-recursive forms. As was pointed out above, there is a definite sense in which recursive schemata are more powerful than non-recursive schemata: this is discussed in [4] and will not be dwelled upon here, for the basic point that we wish to make is that all of the recursion that is used in the evaluation of a semantic parse is done by the uninterpreted functions. The parse can be analyzed into a non-recursive program schema. To see this, simply note that each parse represents a sequence of assignment statements and transformations on this sequence. The transformations involve looping and conditionals, but are non-recursive in character, or at least, have been non-recursive in every semantic computation that we have dealt with in our

work. Of course, there is no formal proof as yet that this must be true of natural language, and indeed, this seems to be an empirical assumption about natural language that is borne out by our implementation. Since we want the non-transformational schemata to form a subclass of the transformational schemata, this restriction is also applied to them.

The second restriction is the one that is applied to the class of transformational schemata to obtain the non-transformational schemata. The essential idea, as we have already mentioned, is that a non-transformational schema is a transformational schema that consists only of assignment statements, followed by a single return. We shall attempt to show that this is an adequate analysis of non-transformational schemata, by giving an algorithm using such a schema, that allows one to traverse a tree from leaves to root in end-order. It should be pointed out that this is not the usual end-order traversal algorithm of computer science, for we start at the leaves of the tree, rather than having to find those nodes. It should be intuitively obvious that such a traversal corresponds to a non-transformational evaluation of a semantic parse tree, for all this traversal does is to start at the leaves of the tree and then on the basis of the denotation of each leaf compute the denotations of the immediate predecessors of the leaves, and then in turn compute the values of the nodes above those, and so on, until finally the value of the root has been computed. But this is just what we mean by a non-transformational

computation: the semantic functions are applied to the denotations of the terminal nodes of the parse tree to compute the denotations of the nodes immediately above them, and then in turn, the semantic functions associated with the derivation of these nodes are applied to these new denotations to obtain the denotations of successively higher nodes in the tree until finally the denotation of the root node has been computed. We shall present the traversal algorithm only for binary trees, but the generalization should be fairly obvious.

Algorithm: Let T be a binary tree with nodes n1, n2, n3,...,nm where nodes np,...,nm are leaves of the tree, and n1 is the root of T and where associated with each non-leaf node of the tree there are a functions f1, f2,...,fp-1 such for $0 < i < p$, we have that fi is a function of precisely as many arguments as ni has successors in T. Assume that there are functions left and right which when applied to a node of T return, respectively, the left successor of the node and the right successor of that node. If the appropriate successor is missing then the value NIL is returned by the function. Then the following non-transformational schema returns the value of T if the values of vnp,...,vnm are set at the beginning of the computation to the denotations of np,...,nm, respectively. The value of T is the value of vn1.

```
(block
(vp-1 <- (fp-1 (left p-1) (right p-1)))
(vp-2 <- (fp-2 (left p-2) (right p-2)))
                ....
   (v1 <- (f1 (left 1) (right 1)))
          (return v1)).
```

In order to clarify this a bit, we shall give an example: Figure 1 shows a simple binary tree, the value of whose top node we wish to compute. We shall refer to the leaf nodes as x4, x5, x6, and x7, and assume that there are functions associated with nodes 1, 2 and 3. Using the algorithm, we get the following schema.

```
(f x4 x5 x6 x7) = (block
   (x3 <- (f3 x6 x7))
   (x2 <- (f2 x4 x5))
   (x1 <- (f1 x2 x3))
      (return x1)).
```

It should be fairly clear that this schema represents a straightforward bottom to top computation of the value of the root from the values of the terminals.

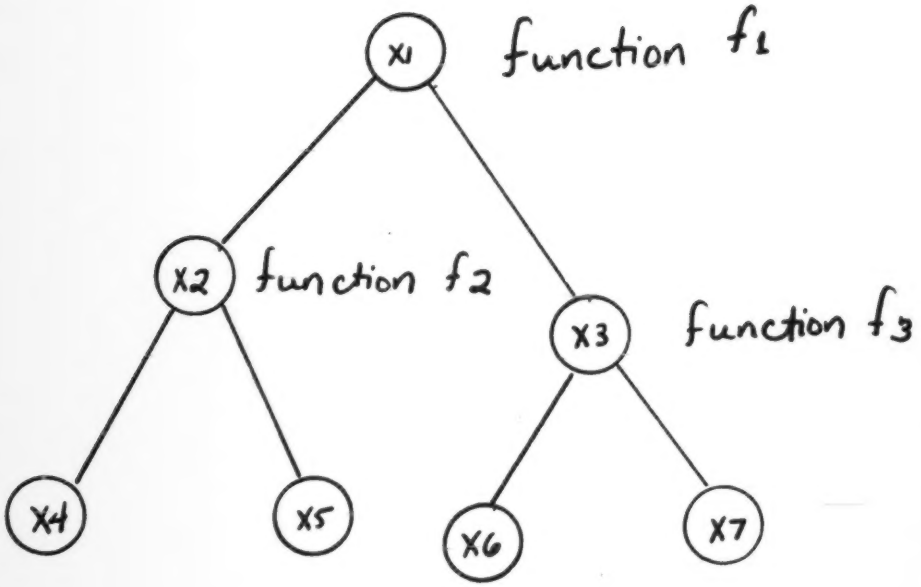


Figure 1. Simple Binary Tree for Evaluation

II.10 The Relative Power of the Non-Transformational and Transformational Schemata

We now wish to show a rather elementary fact about the classes of schemata that we defined in Section II.8. This fact involves a notion of equivalence of schemata. The idea is that there is a transformational schema which cannot be programmed using only the computational structures available in the class of non-transformational schemata.

Definition: Let S1 and S2 be two schemata. Then S1 is equivalent to S2 if and only if S1 and S2 either both fail to terminate or return the same value for all interpretations of the primitive function letters.

This definition is due to Hewitt [4]. We shall denote the function computed by a schema S under some fixed interpretation by fS.

Theorem: There is a transformational schema that is not equivalent to any non-transformational schema.

Proof: We must show that there is some transformational schema S such that for any non-transformational schema N there is some interpretation I such that fS is not equal to fN on some input in the domain of the computation. Let S be the following schema.

```
(g x y) = (repeat (z <- x)
(if (P y) then (return z))
          (z <- (R z))
          (y <- (L y)))
```

Consider the following interpretation. Let the domain of the

interpretation be the set of natural numbers, and let P be a test for equality with 0, let R be the successor function, and let L be the predecessor function. Interpret all other function letters as constantly zero functions. Furthermore, assume that initially all variables except the input variables x and y are set to 0. It should be evident that under this interpretation fS is the ordinary addition function for the natural numbers.

We claim that under this interpretation, there is no equivalent non-transformational schema. Let N be a non-transformational schema, and assume that N has m statements, where following Hewitt, we define the number of statements in a schema to be the number of left parentheses in the schema. By the interpretation of the function letters, we know that the execution of any one statement can add at most one to the value of any variable in the schema. (Recall that any function call in a non-transformational schema has only identifiers as its arguments, so that nested function calls in one assignment statement are not allowed). But since in a non-transformational schema each statement can be executed only once, the value returned by N can be at most the maximum of the initial values of x and y plus m. If we take the input value of x to be 2m and that of y to be 3m, then we have that at best N can return 4m, which is less than the value 5m returned by S for these inputs. Hence, N and S are not equivalent. But N was an arbitrary non-transformational schema, so that the result is proved.

Clearly, there are many other formal results about schemata

that may have application to computational linguistics. One important class of theorems in this area would be results that would indicate some natural restrictions upon the computational power of our transformations. The lack of such constraints has always been the great difficulty with syntactic transformations, but by using schemata, we can hope to obtain some reasonable restrictions upon the computational power of semantic transformations. The discovery of such results is an important open problem.

II.11 A Final Example

Before going on to the discussion of the actual implementation, we shall give an example of how one would represent an actual semantic computation in terms of schemata. The semantic parse that we use is the one that was discussed earlier for the question

Is 2 or 3 even?

As we stated previously, this is parsed into

(S (CHL (LST 2) (LST 3)) (STS EVEN)).

In Figure 2 we have a graphical representation of the surface structure and deep structure trees for this sentence.

The schematological representation for the computation needed to evaluate this is as follows. Let x1 be the list of choices, let x2 be the set of even numbers, and let us have the following interpretations of the functions:

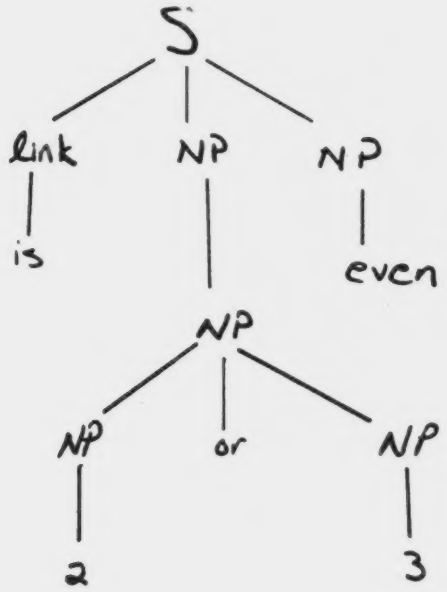
S1 checks for an empty list (NULL)
S2 adds an item to the front of a list (CONS)
S3 is the subset function

S4 selects the first element of a list (CAR)
S5 selects the rest of a list (CDR).

The constant NIL is to represent the empty list. The schema that represents the desired computation is:

```
(V x1 x2) = (repeat (z <- NIL)
 (if (S1 x1) then (return z))
 (z <- (S2 z (S3 (S4 x1) x2)))
       (x1 <- (S5 x1)).
```

SURFACE STRUCTURE



DEEP STRUCTURE

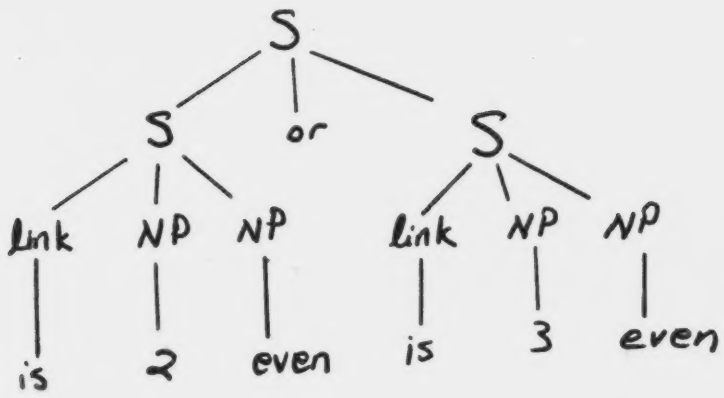


Figure 2. Surface and Deep Structure Trees for a Sample Sentence

Chapter III

Aspects of the Actual Implementation

In this chapter we shall discuss the actual computer implementation of the semantic evaluator. There are several issues to be discussed in this area, and we have several proposals relating to future implementations. In particular, we shall discuss the computer language features that would be desirable for implementations of this type of system. However, for the sake of completeness, we shall present in Section III.1 a brief discussion of the programming language LISP, and also present the notation that we shall use in the rest of this chapter. This will be followed by an analysis of the utility of certain concepts from current computer science: this is the content of Section III.3, Section III.4, Section III.5, and Section III.6. Most of what we have to say here is drawn from the work of others, but it represents the area from which the most important improvements to our work will come. The actual programming that we did is covered in detail beginning in Section III.7.

III.1 Concepts From LISP

The language in which our actual implementation will be described is an extension of LISP called MLISP. In point of fact, we shall need only a small fragment of the language which we present

below. For more details, the reader is advised to consult [14], [15], and [16].

Definition: The BNF definition of MLISP is given below:

```
                <program> ::= <expression>
<expression> ::= <simple expression> [<infix operator>
                 <simple expression>]*
<infix operator> ::= * | + | @ | = | not= | <identifier>
                <prefix> ::= <identifier>
            <simple expression> ::= <block> |
                    <if expression> |
                  <while expression> |
                   <for expression> |
                  <until expression> |
                <assignment expression> |
                    <function call> |
                  <quoted expression> |
                        <atom> |
          <prefix operator> <simple expression> |
                    (<expression>)
                  <block> ::= BEGIN
                 [<declaration> ; ]* |
                  [<expression> ; ]* |
                   <expression> END
         <declaration> ::= NEW <identifier list>
  <identifier list> ::= <identifier> [<identifier>]*
           <lambda expression> ::= LAMBDA
          (<identifier list>); <expression>
        <if expression> ::= IF <expression>
       THEN <expression> [ELSE <expression>]*
          <for expression> ::= <for clause>
            (DO | COLLECT) <expression>
   <for clause> ::= FOR [NEW] <identifier> (IN | ON)
                    <expression>
        <assignment> ::= <regular assignment>
<regular assignment> ::= <identifier> <- <expression>
 <function call> ::= <identifier> (<argument list>)
 <argument list> ::= <expression> [<expression>]* |
                     <empty>
       <quoted expression> ::= '<s expression>
            <s expression> ::= <atom> |
                               () |
        (<s expression> . <s expression>) |
       (<s expression> [[,] <s expression>]*)
         <atom> ::= <identifier> | <number>
  <identifier> ::= <letter> [<letter> | <digit>]*
```

We shall now present some of the basic LISP functions that will be used in the discussions of the actual implementation. First we need to highlight the definition of an s-expression that was given above.

Definition: An s-expression is a LISP atom or is

(s1 . s2)

where s1 and s2 are s-expressions.

The primitive functions of LISP can be explained in terms of this notion. There are two functions that analyze s-expressions into their parts--CAR and CDR. We shall assume that both of these are undefined for atoms, and that

CAR ((s1 . s2)) = s1

and

CDR ((s1 . s2)) = s2.

There is also a LISP primitive, called CONS, used to build more complex s-expressions from simpler ones.

Definition: If s1 and s2 are s-expressions, then
CONS (s1, s2) = (s1 . s2).

We shall now define the notion of a list: in general, the only s-expressions that we shall deal with will be lists. The empty list, which is called NIL, is distinguished by the LISP system. NIL is both a list and an atom to LISP, and is also used to represent the truth value false: anything that is non-NIL is considered to be true.

Definition:
(1) NIL is a list.
(2) If l1 is a list and s1 is any s-expression, then
CONS (s1, l1) is a list.

Lists are written as sequences of elements surrounded by

parentheses. The s-expressions that appear in a list are called the members of the list. There are some other LISP functions that we shall use later. Among these are NULL, ATOM, NUMBERP, and LIST. The LIST function is LISP function of an indefinite number of arguments that returns as its value a list whose members are the arguments in the call to LIST. The NUMBERP function tests whether its argument is a number: it returns T if so and NIL otherwise. The ATOM function tests to determine whether its argument is an atom or not. It returns T if so and NIL if not. Finally, NULL is a function that accepts one argument and tests that for equality to NIL. If it is equal, then the function returns T, and in all other cases it returns NIL.

There are also some convenient ways in MLISP to express iterative constructions: as some of these are used in our subsequent discussions, we shall go over them now. The basic iterative construction that we shall use is the FOR expression. The FOR expression is a command to the computer to step through a list and to perform a certain operation upon each element of the list. If the FOR expression uses COLLECT mode, then the results of each of these operations are assembled into a list and returned as the value of the expression while if the DO mode is used, then the value returned is the value of the operation on the last element of the list. MLISP has many other iterative constructions which are described in [14], but which will not be used here.

Although we shall not deal with the evaluation algorithm that

is used by the LISP system in any detail until Section III.4, we do need to introduce some terminology that will be used throughout this chapter. In general LISP functions evaluate their arguments, but there is a special class of them which do not. A standard LISP function that evaluates its arguments before using them is called an EXPR while a function that does not have its arguments evaluated prior to being called is known as a FEXPR.

III.2 Types of Functions to be Implemented

As we pointed out in the Section II.4, there are two different types of functions that are to be implemented. One of these is the type of function that appears in the surface structure of the input sentence: examples of these are the purely mathematical functions such as factor and multiple. Another type of function is that which corresponds to the English input, but that does not appear explicitly in the surface structure. These are not mathematical functions in the sense that factor or divisible are. Moreover, many of these functions have a transformational character in that they control the sequence of evaluation by the values that they return.

Some simple concepts from computer science are needed to understand how the implementation works. The semantic evaluator is written in Stanford AI LISP 1.6 although it could be modified to run in any other LISP dialect. Since the evaluation program is in LISP, the semantic functions are really functions, for every program in LISP

returns a value. This applies to both the mathematical functions and those functions that serve to implement the control structures that relate functions to their arguments (Section II.4).

III.3 Side Effects

There are several other aspects of the programming language LISP that are important to the understanding of our implementation. One of these is simply that functions may have side-effects. That is to say that while the function returns a value, it may also modify the list structure that the LISP interpreter maintains. Although the mathematical theory of computation at present lacks the tools to give a complete analysis of functions that have side-effects, we shall attempt to give both an intuitive idea of what we use these side-effects for in our system and also a reasonable mathematical analysis of them. The mathematical analysis will, of necessity, be somewhat informal and incomplete due to the difficulty of the problem.

The first thing that ought to be pointed out about the use of functions with side-effects in our system is that we use them primarily to define functions which are not part of the actual LISP code, but which are derived from programmed functions and which represent intermediate steps in the semantic evaluation procedure. Of course, this has profound implications (which are discussed in Section III.14 for the structure of our system, but let us for the present delve into how all of this actually operates.

In LISP there is a systematic attempt to avoid making a distinction between programs and data. In the implementation that is actually used in this project, the function definitions are simply special types of list structures that may be manipulated by the LISP system in the usual ways. These list structures have the additional property that they may be applied as functions to other list structures. Thus, we can in the course of evaluating a semantic parse produced by the CONSTRUCT program create functions that are not in the pre-coded semantic evaluation program and use these in the later evaluation of the semantic parse: the appropriate list structures are simply created and stored. This is done, for example, in the manipulation of the characteristic functions of sets. To handle the intersection of the set of odd numbers with the set of prime numbers, the system produces a new characteristic function for the set of odd prime numbers by combining the characteristic functions for the set of primes and for the set of odd numbers. Some of the details of these created functions will be described in Section III.10. For now it is sufficient to note that the side-effects that are used by the semantics system are rather limited in nature and rather straightforward: there are no functions which make massive and strange changes to the list structure. All of the functions that are defined are based on rather simple combinations of functions that previously existed in the system and involve only locally available list structure, i.e., things that appear as arguments to the functions that create new functions from old.

In an attempt to make this a little more precise, we shall go into some more detail about the structure of functions with side-effects and their properties. What we shall say in the following is based on the concepts of Section II.8 and is not entirely accurate for any actual LISP system, but it does convey in some ways an abstract picture of what is going on. What is important here is not the lists maintained by the interpreter itself, but rather the functions and structures that are accessible to the user.

Definition: Let f be a transformational schema described in the notation of Section II.8, and let x1, x2, x3,..., xn be the input variables of f. Assume that f returns its value in a variable z. Then f is said to have a side-effect if and only if there exists y, a variable distinct from z, such that the value of y is changed by the application of f to x1, x2,...,xn.

Definition: Let f be as above, and let y1, y2,...,ym be the only variables in f other than x1, x2,...,xn. Then f is said to have only local side-effects if and only if the only variables that are altered by the application of f to x1, x2,...,xn other than z are among y1,..., ym.

Our fundamental claim is that other than for the lists that are maintained by the LISP system the functions that are used by our semantic evaluator have only local side-effects. In order to verify this claim, we shall consider the role of side-effects in the class of transformational schemata. Any non-local side-effects that may be

produced by the LISP system are the result of the actual application of a particular instance of this schema to some arguments, and should, if the LISP interpreter is properly written have no effect on our functions. Hence, these schemata are a good model for our semantic evaluator.

Theorem: A transformational schema S can have only local side-effects.

Proof: This is actually quite obvious, but we shall explain what is going on at the risk of belaboring the point. Let S be a transformational schema. Then we shall show that S has only local side-effects. The only constructions in a transformational schema that can affect the value of a variable (whether it appears in the S or not) are the assignment statement and the return statement. But both of these constructions must contain the name of the variable whose value is to be changed. Hence, the variable appears in S.

A formal proof of this would involve an extremely messy induction on the structure of transformational schemata or on the length of S.

III.4 The LISP Calling Sequence

However, the LISP interpreter has some features which are actually in the way of doing a proper implementation of the semantics system. The most important of these is the LISP calling sequence or evaluation algorithm. As is well known, LISP uses a recursive inside-

out method to evaluate functions. That is, the LISP interpreter first checks to see if the function being called is a function that evaluates its arguments. If this is the case, then the arguments are evaluated before any evaluation of the function itself. Note that the evaluation of the arguments to the function may involve the evaluation of other function calls--including perhaps calls to the same function that appears at the top level. Once the evaluation of the arguments is complete, the function is then called on those arguments. If the function does not evaluate its arguments, then it is immediately called on its apparent arguments. Any function or any instance of functional application is an s-expression. The LISP function that actually governs the evaluation of s-expressions is called EVAL. EVAL may be given the following recursive definition for functions that evaluate their arguments.

EVAL (X) <- IF ATOM X THEN VALUE X ELSE
CAR X (CDR X)

where VALUE x is a special value that is associated with the atom and where in the other case, we apply the CAR of the expression to the CDR.

While the recursive inside-out algorithm described above is the one used by LISP to evaluate our functions, the functions themselves are coded in various strange ways to prevent the LISP interpreter from really doing the evaluation in exactly that order. The basic method of controlling the calling sequence in the semantic evaluation routine involves the introduction of certain special functions into the semantic routines that serve as flags to the LISP interpreter that tell

it not to evaluate the s-expression in question. These flags are written as LISP functions and appear as the CAR of an expression. LISP sees the function and applies it to the CDR of the s-expression--usually doing nothing to that part of the expression. Intuitively the function often also serves as a data type telling outer functions what type of arguments that they have received and hence how to behave on these data.

To clarify this point further, let us give an example of one of these functions: this example will be given in somewhat general terms as we shall cover the actual function in Section III.7. The function LST is defined in such a way that it passes an explicit list of elements to higher level (outer) functions. LST evaluates non-atomic expressions that occur as members of its argument list: it always takes a single list as its argument. Atomic members of the list of arguments are not evaluated, but rather are passed up to the next level of the s-expression. Since EVAL normally would evaluate everything in the list before calling LST, LST is defined to LISP as a function that does not want its arguments evaluated. When LST is called it scans its arguments and decides which ones it wants to evaluate.

Another type of function that appears in the semantic evaluation routines is that which serves to alter the calling sequence of LISP to produce the effect of a transformation. As was pointed out in the previous chapter, our linguistic system obtains its power from

this feature. There are essentially two different reasons why such a transformation is necessary. The first is the infinity problem--one cannot list infinite sets. The other is that the semantics--the intuitive meaning of the sentence--demands that the order of evaluation be based on something other than proximity within the surface sentence. One of the most surprising things about the semantic evaluation routines is that these two very different problems can be handled in essentially the same way. In both cases LISP performs an evaluation that has a side-effect, the side-effect being to define a new LISP function that is later applied to arguments. For example, if we are asked to apply a set-theoretical intersection function (the I function) to two sets that we have no reason to believe to be finite, then the characteristic functions of the sets are combined into a new characteristic function for the set which is the result of the operation upon the two given sets. Similarly, if we are given something that we cannot really evaluate in the current context, then certain flags may be set and the whole thing passed to the next level.

III.5 Backtracking

The standard computer science method of dealing with this calling sequence problem is, of course, backtracking--something that is absent in LISP although present in languages such as PLANNER and MLISP2. In order to explain clearly how our implementation differs from this standard we shall present an informal account of backtracking or nondeterministic algorithms which is drawn from Floyd [17].

Nondeterministic algorithms are simply standard algorithms expressed in some suitable language such as that of Section II.8 with the exception of the introduction of a multiple-valued interpreted function CHOICE whose values are less than or equal to the the value of its argument. The idea behind CHOICE is that one makes an arbitrary decision at the choicepoint of which branch to take. It is known that for the class of recursive functions this does not add any power although it is the case that for pushdown automata one does gain some power by the use of such a function. Following the treatment of Floyd, we shall also assume that each of the terminal points of the schema is labelled by either SUCCESS or FAILURE. Only those computations that end at nodes labelled SUCCESS are considered to be computations using the schema. Again this feature does not add any power to programming language in the sense that no more functions will be computed than without this than with it. However, the fact that recursive function theory does not make any discrimination in this regard is probably more of a failing on its part as a mathematical theory of computation rather a lack of any real difference. Moreover, it is shown in [4] that schemata that allow certain types of backtracking are more powerful than transformational schemata in the sense that there is a backtracking schema that is not equivalent to any transformational schema. While we have not explored the relationship between the kind of backtracking described here and that of Hewitt, it seems quite likely that a similar result can be obtained using our formulation.

As we stated above, our system does not use any backtracking techniques. We regard this as an advantage at its current state of development for the simple reason that the use of relatively weak methods at this point means that with more powerful techniques it should be possible to extend the system fairly easily.

III.6 Interrupts and Demons

Another feature of computational linguistics systems such as that of Winograd is the use of asynchronous processing via the use of interrupts and demons. (For more discussion of the work of Winograd, see Section V.3.) Indeed, Winograd gets the transformational power in his system from the use of a demon in his parsing routine: this is how, for example, he checks for agreement. In our relatively simple semantic evaluation system this feature is absent. All of our transformational power is gotten from the use of functions that delay their actual evaluation to the appropriate time. It seems to be rather clear at this point that much of our system could be coded somewhat more easily by the use asynchronous processing features, for rather than having to trick the LISP interpreter into changing the calling sequence for various functions, we could simply use the asynchronous processing feature to generate an interrupt to handle any processing that is desired but not in the standard recursive inside-out order. What is lost by doing this is a certain amount of conceptual clarity, for in event-directed programming it is often difficult to discover in

what order things are actually evaluated when the program is actually run. Furthermore, the analysis of such programming is quite difficult, and at least at the present time we are not satisfied that the current ideas of multi-process schemata (see .4.) are adequate to deal with this computational structure. In our system this can be discovered quite easily by simply looking at the definitions of the functions in the semantic parse of the input sentence.

III.7 The Implementation of Data Typing

At this point we are in a position to give a detailed description of our actual implementation. The first thing that will be described is our method of data typing--how it works and what it is good for. Then we shall deal with the functions that correspond to the mathematics of the system all of which appear in the surface structure of the input sentences, and all of which are quite straightforward in the mathematical sense as well as in their implementations. Finally, we shall deal with the functions that correspond to the English of the system; these are the functions that perform transformations to the semantic parse tree.

In order to understand the manner in which the type functions operate it is necessary to explain something about the way in the CONSTRUCT program processes its input and the kinds of things that are passed to the semantic evaluator. Essentially, the CONSTRUCT program produces a semantic parse which is in the format of an s-expression.

This is passed directly to the LISP READ program, which in turn reads it and calls the toplevel of LISP, EVAL. The CONSTRUCT program is written in such a way that each atom in the s-expression that is passed to LISP is quoted: this can be done in AILISP by placing an atsign in front of each atom. What LISP sees as atoms are strings to the CONSTRUCT program, which is written in SAIL. The atoms in the semantic parse correspond to words of the input sentence and calls to transformational functions (obligatory semantic transformations if one likes that terminology). It should be noted that the quotation marks, which are actually calls to the LISP QUOTE function, are present primarily for historical reasons: the current type functions serve to eliminate the need for them. Indeed, the quote functions are a bit of a nuisance as the type functions must make special provision to insure that EVAL is called on those quoted atoms which are used to index into the data base that the program uses. The reason is simply that the atom and the quoted atom are seen as different structures by the functions that are used to index the database. By calling EVAL on the quoted atom, the quotation function is removed from the list structure, and the atom is seen by LISP as the atom itself. An example should make this a little less opaque. Suppose that A is a LISP atom. Then A has a list of elements associated with called its property list: data can be stored on this property list under various property names and retrieved with a simple LISP function, called GET. However, if GET is called with (QUOTE A) rather than with A and if GET is told not to

evaluate the expression it receives as the semantic evaluator informs it that it should do, then the wrong property list is scanned and the information that is desired is not found. Thus, it is necessary for the semantic type functions to eliminate the extraneous quote marks on the atoms that appear in the lists that they govern.

There are ten of these semantic typing functions, and they divide semantically and computationally into two major groups. The first of these is the sentence typing group: these functions appear only at the beginning of a semantic parse and serve solely to tell the semantic evaluator the type of sentence--question, command, declarative, or formula--the input string was. Not too surprisingly, the current evaluation routine has relatively little use for this information because it is not handling declaratives in a realistic way as yet. These sentence types are also for the benefit of the output routines for the system, which are also as yet non-existent. There is one sentence typing function for each of the sentence types indicated above. Each of these sentence typing functions simply evaluates the contents of the argument list and then CONSes on the appropriate sentence type. The following diagram gives these functions and their definitions.

```
DCL DECLARATIVE DCL(X) <- CONS ('DCL, EVAL (X))
QUS QUESTION QUS(X) <- CONS ('QUS, EVAL (X))
FML FORMULA FML(X) <- CONS ('FML, EVAL (X))
CMD COMMAND CMD(X) <- CONS ('CMD, EVAL (X))
```

The other type functions are more complex and more important to

the operation of our program. Essentially all of these except CHL are quotation functions that turn off the LISP evaluation process at that point at which they are encountered. All of these functions share the rather special property that LISP does not evaluate their arguments, and moreover, they all accept an indefinite number of arguments. This is done by defining these functions in a special way. Whenever, evaluation is desired, EVAL is called explicitly. These functions also are used by other functions that are called later as flags: on this point see Section III.10. In some cases the function also simplifies its input: this is crucial for things that are to be seen as explicit lists, for we want to have all of the members of the list in simplest form with any function calls occurring within the list evaluated out. As a result of the quotation marks this is also important for functions which are to be applied, for as was remarked before, these depend upon a database lookup, which in turn depends on the function name being passed to the GET function rather than a quoted version of the function name.

The LST function is defined as follows:

```
LST (X) <- CONS ('LST, FOR NEW I ON X COLLECT
LSTEVAL I)

LSTEVAL (X) <- IF ATOM X THEN X ELSE EVAL X.
```

As an example, consider the following form:

(LST A (PLUS 2 3) (LST B C))

First the A is scanned and found to be an atom. Then the form (PLUS 2

3) is evaluated to 5. Finally, there is the s-expression (LST B C), which involves a call to the LST function again. This time both of the members of the list of arguments are atoms, so that this evaluates immediately to (LST B C). Hence, the value of (LST A (PLUS 2 3) (LST B C)) is (LST A 5 (lst B C)).

There are four other functions whose behavior is quite similar to LST that serve to flag different datatypes. FCN marks a mathematical function like factor; it also serves to eliminate unwanted quotation marks. The function STS flags a set that is represented as a characteristic function while TV marks a truth value. Finally, there is UNT which marks a unit and which is not used in the present implementation. For the sake of completeness, we shall give the definitions of these functions.

```
FCN (X) <- CONS ('FCN, IF CAAR X = 'QUOTE THEN CDAR X ELSE X).
STS (X) <- CONS ('STS, X).
TV (X) <- CONS ('TV, LIST X).
UNT (X) <- CONS ('UNT, X).
```

The CHL function that was mentioned above has a slightly different role from the other type-checking functions. Although it behaves like LST in calling EVAL on the members of the list that follow it, it also serves as a signal to higher-level functions that are applied to it that the list is a special type of list--a list of choices one or more of which is to chosen. For example, the semantic parser produces this type of a list when it parses the list of answers to a multiple choice question. Another example is the sentence

Is 3 greater than, less than, or equal to 2+3?

In this sentence, the CHL, acting as a flag to the higher level functions that serves to change the calling sequence. In this instance, rather than attempting to evaluate together the elements of the phrase "greater than, less than, or equal to" and failing due to the fact that these functions are arithmetical relations that expect numerical arguments, the elements of the phrase are made into a list of choices and the function CHL is applied to this list. At a higher level after the arguments have been seen by the semantic evaluator, the functions that check the arithmetical relations are actually called, which is, of course, the intuitive content of the sentence. This point is discussed further in Section III.10.

Finally, we shall give the definition of the CHL function. It should be noted that like the functions above it accepts an indefinite number of arguments, and does not evaluate these until explicitly told to do so.

```
CHL (X) <- CONS ('CHL, FOR NEW I IN X COLLECT
CHLEVAL I).

CHLEVAL (X) <- IF ATOM X THEN X ELSE IF CAR X = 'CHL THEN X
ELSE EVAL X.
```

III.8 Mathematical Types

We shall now briefly discuss the implementation of appositions in our system. In elementary mathematical language appositions generally are used to express mathematical type-checking, which is

quite distinct from the internal data typing done by the semantic evaluator. For example, one might well ask what the sum of the numbers 2 and 3 is: the word "numbers" serves in this context to call a special function in the semantics fork which checks to make sure that its argument is a number. Of course in this particular case the system just calls the function NUMBERP. In the more complex case of fractions it checks for the internal list representation of a fraction, which is the LISP atom DIV as the CAR of a list, the CDR of which is a list consisting of the numerator followed by the denominator. In MLISP notation this function, which is called FRACTION and which is fairly representative of these kinds of functions is defined as:

```
FRACTION <= IF CAR X NOT= DIV THEN NIL
ELSE IF LENGTH X NOT= 3 THEN NIL
ELSE IF NOT NUMBERP CADR X THEN NIL
ELSE IF NOT NUMBERP CADDR X THEN NIL
ELSE T.
```

III.9 Arithmetical Relations

Next we consider the arithmetical relations, which are the only mathematical functions that are not completely straightforward in their implementation. The problem that is involved here is fairly simple: the semantic parses that are produced are such that these arithmetical relations are not always passed the same number of arguments each time that they are called, but they must never the less evaluate their arguments. Moreover, they must be used in conjunction with the APPLY

function of LISP which does not work properly with LISP FEXPRs. The solution is rather simple: the toplevel function is a FEXPR, but it calls--on the basis of the input that it receives--one of two EXPRs. It is assumed that each relation can only be called with one or two arguments. These auxiliary functions do the obvious thing that the the toplevel function's name implies. The problem with the LISP APPLY function is solved by checking to see if the APPLY function is being called on a FEXPR; if so, then rather than calling APPLY the name of the function is CONSed onto the argument list and EVAL is called on the resulting s-expression. As an example of this type of function, we shall give our definition of the equality relation (the function EQL).

```
EQL <= IF LENGTH X = 1 THEN LIST(LST, CAR EQL1 X)
ELSE IF LENGTH X = 2 THEN LIST (TV, EQL2 (CAR X, CADR X))
ELSE ERROR.
```

As one might expect, EQL1 generates a list of one element--the element with which one is supposed to determine equality or inequality. The MAKESET function that is mentioned below simply removes duplicate elements from a list, i.e., makes the list into a set. The function EQL2 takes two arguments. If both of the arguments are atoms, then the LISP EQ function is called while if both are lists then a special function that compares lists for being equal when considered as sets is called. Note that two sets a and b are said to be equal just in case that x is member of a if and only if x is a member of b: this definition of equality is different from that used by the LISP EQUAL function which checks for equality of list structure. The other

arithmetical relations used in the system are greater than, denoted by GT, less than, denoted by LT, greater than or equal to, denoted by GE, and less than or equal to, denoted by LE.

Almost the only thing that will be said about the actual arithmetical functions themselves is that they are in the system, that they are standard, that they are not too intelligent, and that they are limited to some very common arithmetical operations. Clearly, a better job could be done on these, but that is not really the topic of interest here.

III.10 The Set Theoretical Functions: The I Function

Next we shall consider the heart of the system--the functions that correspond to the English structure of the sentence. Many of these are functions that perform transformations in our system, and many of them represent tricks played upon the LISP interpreter.

As fairly representative of a top level function of the system, i.e., one that appears at the level immediately below the level of the sentence type, is the I function. This function performs a type of intersection, but its exact action depends on the data types that it receives on its arguments. In order to describe this function, we shall first present an abstracted and somewhat simplified version of its definition in terms of the various possible cases that might occur, and then detail what is to be done in each of the cases. In our usual MLISP notation the definition of I is:

```
I (X, Y) <= IF CAR X = 'CHL OR CAR Y = 'CHL THEN ICHL (X, Y)
```

```
ELSE IF CAR X = 'LST AND CAR Y = 'LST THEN ILST (X, Y)
ELSE IF CAR X = 'STS AND CAR Y = 'STS THEN ISTS (X, Y)
ELSE IF CAR X = 'LST AND CAR Y = 'STS THEN ILSTSTS (X, Y)
ELSE IF CAR X = 'STS AND CAR Y = 'LST THEN ILSTSTS (X, Y)
ELSE IOTHER (X, Y).
```

We shall not describe the IOTHER function as that case really does not arise in practice. In order to describe more easily the other functions that are used, we shall assume that the type-checking information is processed by some transparent intermediate functions, so that the type flags (e.g., CHL) have been removed by the time that the arguments are passed to the auxiliary functions mentioned above. It will also be assumed that the functions that we shall discuss below know what those types are. Of course this is not very realistic, and in the actual implementation the higher level functions are organized in such a way as to handle this matter.

The first auxiliary function that we shall describe is the one that handles the occurrence of the CHL flag in either one of the arguments to the I function. The definition of ICHL is as follows:

```
ICHL (X, Y) <= IF CHOICELIST X THEN FOR NEW I IN X COLLECT
I (CAR X, Y) ELSE IF CHOICELIST Y THEN FOR NEW I IN Y COLLECT
I (X, CAR Y) ELSE ERROR.
```

What this means is that for lists of choices the intersection is taken for each choice. This implements the transformation that was discussed earlier. It should also be noted that the definition of the function is recursive in the sense that the top level I function is

called by the function ICHL. This is not really a feature that is actually implemented, but it represents something that is desirable from a conceptual point of view. Now we consider the function ILST: this takes the intersection of two explicit lists--each marked with the LST function. The definition of this function is:

```
ILST (X, Y) <= IF NULL X THEN NIL ELSE IF MEMBER (CAR X, Y) THEN
CONS (CAR X, ILST (CDR X, Y) ELSE ILST (CDR X, Y).
```

This is a very standard function that finds the intersection of two explicitly listed sets. Unlike this function, the function ISTS is somewhat non-standard if very simple to understand once written. The idea is that the input to this function consists of two characteristic functions for the membership relations of the sets of which we wish to take the intersection. Now the characteristic function for the intersection of two sets is just the function that is formed by taking the logical and of the characteristic functions of the input sets. Given this, all that must be done is to convince LISP to do that for us. However, LISP systematically treats functions and data uniformly, so that we may define a function that operates on previously existing function definitions to produce new function definitions: this is just what the ISTS function does.

```
ISTS (X, Y) <= DEFINEFUNCTION ('AND, X, Y)
```

where DEFINEFUNCTION is a function of three arguments that takes the input characteristic functions and stores the definition of a new function on the property list of a new atom. The and means that the new function has the general form

IF X AND Y THEN T ELSE NIL.

The exact details of how this is done in LISP are of some interest here. As we have mentioned before, the basic idea used in handling operations on sets that are represented as characteristic functions is that of the run-time created procedure. Using the characteristic functions of the input sets, a new characteristic function is manufactured at execution time by the semantic evaluator. The set theoretical operation causes the system to first generate a new LISP atom upon whose property list the function definition is to be put. In general for ISTS the format of this function definition is:
LAMBDA (!X) IF (X !X) AND (Y !X) THEN T ELSE NIL.

The variable !X is a dummy variable used in the function definition. It is assumed that X and Y are input characteristic functions, which are to be called by the system during the course of evaluating this form.

This process does have its disadvantages from the aspect of the implementation, for function definitions that are created in this manner remain present within the LISP system for the life of the program. This means that after a time the LISP system will run out of available space, and the normal procedures for collecting space that is no longer needed will fail. Clearly, this flaw in the system can be remedied by means of some systems programming to alter the code of the LISP interpreter.

Finally, there is the case in which the I function receives as its arguments an explicit list and a characteristic function for a set. The idea here is to apply the function to the elements of the explicit list one at a time to check to see which of the members of the explicit list belong to the other set. This is defined by:

```
ILSTSTS (X, Y) <= IF NULL Y THEN NIL ELSE IF X (CAR Y) THEN
CONS (CAR Y, ILSTSTS (X, CDR Y)) ELSE ILSTSTS (X, CDR Y).
```

We have assumed here that the explicit list is known to be the second argument and the characteristic function the first. It should be pointed out that the actual system has an opportunity to call some heuristics before actually going to these procedures. These heuristics incoporate some of the very elementary facts about intersection such as the fact that anything intersected with the empty set is empty and the idempotency of intersection. A detailed discussion of the heuristics themselves is to be found in Section IV.6 and subsequent sections.

III.11 The Other Set Theoretical Functions

There are three other functions that appear on our system that have the same character as the I function except that they compute other set-theoretical functions. These functions are the U, S, and SD functions. The only difference between these functions and the I function is that these functions compute different set-theoretical functions. As should be obvious, the U function computes the union of two sets while the S function determines whether the first set is a a

subset of the second. The SD function determines the set-difference of its first argument and its second, i.e., it determines the set whose members belong to the first argument and not to the second. Like the I function all of these functions look for the CHL flag and perform multiple operations if it is seen. They also use the type information to determine the methods of computation that are to be used. Since there are some special problems associated with each of these functions, we shall briefly discuss of each of them below.

The trouble with the U function is that it is an expansion of the set: there is the simple fact that the union of two sets is a superset of both. The problem with this is that our general program of reducing the representation of sets to explicit lists is set back by the use of the U function: when one takes the union of a set represented by a list with one represented by a characteristic function, one in general cannot represent the result by a list as one can when taking the intersection. In general then the output of the U function may be described as follows:

```
U (X, Y) <= IF LST (X) AND LST (Y) THEN ULST (X, Y)
ELSE IF LST (X) AND STS (Y) THEN ULSTSTS (X, Y)
ELSE IF STS (X) AND LST (Y) THEN ULSTSTS (X, Y)
ELSE IF STS (X) AND STS (Y) THEN USTS (X, Y)
ELSE UOTHER (X, Y)
```

As before the case of UOTHER does not arise in practice. The definitions of ULST, ULSTSTS, and USTS are given below:

```
ULST (X, Y) <= IF NULL X THEN Y ELSE IF CAR X MEM Y
THEN ULST (CDR X, Y) ELSE CONS (CAR X, ULST (CDR X, Y))

USTS (X, Y) <= DEFINEFUNCTION (W, MEM X OR MEM Y)
```

ULSTSTS (X, Y) <= DEFINEFUNCTION (W, MEM X OR MEM Y)

where we have DEFINEFUNCTION as usual defined to be a function that creates function definitions inside LISP. The function that is defined by a call to this s-expression simply checks for membership in X and Y by scanning the argument if it is a list and by applying the characteristic function otherwise.

The S function is rather different from the other functions in this group because it does not return a set, but rather a truth value. The only output from the S function has the form (TV T) or (TV NIL). It should be pointed out that the S function is especially hard to compute for sets that are not expressed as explicit lists. The reason is that there is no uniform and effective way to determine if one recursive set is a subset of another: this is a standard result of recursion theory and may be found, for example, in [5]. This means that unless the system is able to use the definitions of the sets, i.e., knows some mathematical relationship between the sets, it cannot compute the S function if both of the arguments are given as characteristic functions. If the first argument is an explicit list, then the algorithm that is used to compute the S function is apply the characteristic function for the second argument to each element of the list. Finally, if the second argument is a list and the first is given as a characteristic function, more information is again needed. The problem this time is that there is no straightforward way to generate all of the members of a set that is given as a characteristic function. The standard definition of the functions is given below.

```
S (X, Y) <= IF LST X AND LST Y THEN S1 (X, Y)
ELSE IF LST X AND STS Y THEN S1 (X, Y)
ELSE SOTHER (X, Y)

S1 (X, Y) <= IF NULL X THEN T
ELSE IF MEM (CAR X, Y) THEN S1 (CDR X, Y)
ELSE NIL
```

The last member of this class of functions is the SD function. This function is more closely related to the I and U functions than to the S function. Like the U and the I functions this function returns a set whose members are the members of its first argument that do not belong to its second: it takes the set difference function. As usual the function is quite straightforward for sets that are represented as explicit lists. Moreover, it is possible to specify the characteristic function of the answer in all of the other cases, but not possible in general to give a representation as an explicit list, so that the SD function resembles U in this regard. The following LISP definitions describe the SD function.

```
SD (X, Y) <= IF LST X AND LST Y THEN SDLST (X, Y)
ELSE IF LST X AND STS Y THEN SDSTS (X, Y)
ELSE IF STS X AND LST Y THEN SDSTS (X, Y)
ELSE IF STS X AND STS Y THEN SDSTS (X, Y)

SDLST (X, Y) <= IF NULL X THEN NIL
ELSE IF CAR X MEM Y THEN SDLST (CDR X, Y)
ELSE CONS (CAR X, SDLST (CDR X, Y))

SDSTS (X, Y) <= DEFINEFUNCTION (W, MEM X AND NOT MEM Y)
```

III.12 The Verb 'Have' in an Existential Context

We now turn to the discussion of the implementation of the verb "have" in our system. As was mentioned in the previous chapter, this verb is quite interesting due to the transformational character of its semantics. Let us briefly recall what we said about the intuitive semantics of "have". One of the fundamental ideas about this fragment of natural language is that sentences in general and questions in particular can be viewed as structures that relate functions to their arguments: the so-called linguistic or English part of the sentence serves to bind functions to their arguments and to ensure that the functions are called in the proper sequence. For the case of questions using the verb "be" this binding is rather like one based upon structural proximity; however, in the case of the verb "have" this is not the case. For example, we have the sentences:

Are the factors of 6 even? (1)
Does 6 have an even factor? (2)

To compute the answer to the first question, we simply apply the factor function to 6 and then check to see if all the members of the resulting list are even numbers. But to answer the second question, we must ignore the propinquity of even and factor, and again apply the factor function to 6 and then check to see if any members of the resulting list are even. This clearly involves a semantic transformation which passes 6 as an argument to the factor function and then applies the characteristic function of the set of even numbers to each of the elements of the resulting list. This aspect of the semantics is implemented using special functions that are introduced

into the semantic parse by the appearance of the word "have" in the surface sentence.

The function that would be used in the computation of the answer to question (2) above is called EXTHNP. The definition of EXTHNP is given below in our standard notation.

```
EXTHNP (X, Y) <= IF LST Y THEN DEFINEFUNCTION (EXP1, X, Y) ELSE
IF STS Y THEN DEFINEFUNCTION (EXP2, X, Y)  ELSE ERROR.
```

In the above EXP1 and EXP2 are templates used in the definition of the temporary functions that EXTHNP generates. Both carry the force of a logical and. Also both of the expressions that are used in defining the function that is generated by EXTHNP call the I function: the difference in EXP1 and EXP2 is that in EXP1 the I function is called for two explicit list while in EXP2 the I function is called for one explicit list and one set. The definitions of EXP1 and EXP2 follow:

```
EXP1(X, Y) <= IF EXIST (I (APPLY X !X) ('LST Y)) THEN T ELSE NIL
EXP2 (X, Y) <= IF EXIST (I (APPLY X !X) ('STS Y)) THEN T ELSE NIL
```

By EXIST we mean a LISP function that checks to see whether its argument is an empty set or not. For explicit lists we have the simple function that is shown below.

```
EXISTLST (X) <= IF NULL X THEN NIL ELSE T
```

For the case of a set which is described by its characteristic function the EXIST function is more difficult to program. The idea that we use is fairly simple--search for an example of something that belongs to

the set. For sets of natural numbers this can be done in a fairly straightforward manner by simple enumeration and the use of a few heuristic tricks; it is also the case that the information extraction procedures discussed in the next chapter are applicable to this problem.

III.13 Two Other Constructions Using 'Have'

We shall consider two other functions that are used to handle English constructions involving the verb "have". The first of these is called UNVHNP, and handles sentences in which we are asked to determine if all of the members of some set have some property or other. For example, consider the question:

Does 12 have only even factors?

In this question we must first determine the set of all factors of 12 and then check to see if each member of this set is even.

The definition of UNVHNP is roughly as follows:

UNVHNP (X, Y) <= DEFINEFUNCTION (W, IF S (X (!X), Y) THEN T ELSE NIL)

In the formula above S is to be taken to be the subset function of Section III.11, and !X is a dummy variable that is used in the definition of the function that UNVHNP creates. For the example given above, we have the following semantic parse:

(QUS (S (LST 12) (UNVHNP (FCN FACTOR) (STS EVEN)))).

The function that UNVHNP creates at execution-time has the form:

IF S (FACTOR (!X), EVEN) THEN T ELSE NIL

The other function that we wish to discuss briefly is called FCNMK. This function is used to take pre-defined arithmetical functions of our system into new functions that are somewhat modified. It accepts two arguments, the first of which is an arithmetical function whose definition is already specified and the second of which is a set of numbers. It creates the characteristic function of the set of all members of the result of the application of its first argument to some number intersected with its second argument. For example, FCNMK would be called to handle the phrase "the odd factors". It creates a function that computes just the odd factors of its argument. As is standard with this kind of function, FCNMK creates a special function that is stored and later used by the semantic system to actually compute, say, odd factors. The definition of FCNMK is given below:

```
FCNMK (X Y) <= DEFINEFUNCTION (W, I (APPLY (X, !X), Y)).
```

This definition follows our usual conventions, and involves the I function that is discussed above. For the example of "odd factors" a function would be created that would generate the odd factors of a number when called:

```
I (APPLY (FACTOR, !X), ODD)
```

is the result of such an application.

III.14 The Importance of the Run-time Creation of Functions

With the description of this function we have completed detailed descriptions of representatives of each of the major types of functions to be found in the semantic evaluator. In the next chapter we shall talk a bit about why this works as well as it seems to: there are some tricks involved. However, one of the major tricks should be apparent already: many of the functions that are needed by the system are not hand-coded in advance, but are created by calls to other functions at run time. This, needless to say, greatly increases the power of our system, for the programmer does not have to pre-code every function that is needed for the semantic evaluation process.

Chapter IV

Information Extraction and Heuristics

In this chapter we shall attempt to account for the fluency of our semantics system and to indicate how its apparent performance as a question-answering system might be improved. We shall present first a method of using the definitions of functions to help answer questions about those functions: we call this information extraction. After this we shall give a rather detailed coverage of the heuristics that are actually used in our present system and some simple extensions of them.

Information extraction is introduced in Section IV.1, and the relationship between it and the standard methodology of resolution theorem-proving is covered in Section IV.2. The most important use of this technique should be in the answering of "how" questions (Section IV.3), but it is envisioned that it will be applicable to other problems as well Section IV.4. The heuristics that we have actually implemented already are the subject of the rest of the chapter (Section IV.6 and onward).

IV.1 Introduction to Information Extraction

At this point in the discussion of the implementation, we shall consider what could be a very important application of pattern-matching

to our system. At present this has not been put into the program, but it seems to hold some promise. The idea is that the characteristic functions of the sets as well as the LISP functions that code the arithmetical functions contain information about the sets or functions that is accessible to a clever program. Such a program could extract information from these procedures which would be useful both in answering questions about why something is done in a certain way or how it is to be done. For example, one might consider the question of whether one divides to find the factors of 6. In order to answer this question, one might, instead of looking in a data base or proving a theorem about the answer, extract this information from the function that actually computes the factors of a number. Similarly, one might be able to tell from the characteristic functions for two sets if their intersection is empty or not. How sophisticated this can be made is dependent primarily on the amount of effort that one wants to put into it. It is fairly clear that for the rather stylized definitions that are created by the I function it would not be too difficult to check to see if, for example, the function that is created is the characteristic function of the numbers greater than 5 and less than 2. This is type of information extraction from procedures is very much like the kind of theorem proving that is done using pattern-matching in the PLANNER system of Hewitt [4] except that the functions that are called by the procedures are compared with each other on the basis of pre-stored knowledge about their mathematical relationships. It should be noted

in this regard that we do not envision the use of the methods of resolution theorem-proving in doing this. Although resolution has been suggested by some authorities such as Sandewall as a basis for question-answering in natural language, there are a number of reasons to believe that this is not really a very desirable approach to the problem of natural language question-answering.

IV.2 Resolution Is Not a Suitable Basis For Information Extraction

There are several reasons why the resolution method does not really serve to provide a good basis for natural language question-answering. The first is a very pragmatic reason, and also, given the state of the current resolution theorem-provers, a very compelling one. At the present time (mid-1973), resolution theorem-provers are too inefficient at finding proofs and require too much central processor time to be practical. Even the best-written theorem-proving programs are very large and rather slow. Although this feature is not particularly bothersome when one is really interested in theorem-proving for the sake of theorem-proving, it makes it impossible to use a theorem-prover as the basis for question-answering in reasonably close to real time. The problem is, of course, that the resolution method is a uniform proof procedure designed to handle many kinds of deductions and is not specialized enough to meet the needs of a question-answering system. Although Sandewall has suggested that this

objection can be met by writing resolution theorem-provers that are more efficient, this has not as yet been done successfullly, and so the objection remains.

There are other reasons for believing that resolution theorem-proving is not a very viable method for answering natural language questions. The first of these, which was pointed out to me by R. Statman, is that resolution is not a very natural method of reasoning. In particular for propositional logic, forming resolvents is rather difficult, and produces proofs that are not as "natural" as one might obtain from a natural deduction calculus. The structure of a resolution proof is rather more complex than the structure of a proof in a natural deduction system. In particular, there is no very clear relationship between the formulas at the top of the resolution proof tree, and the end formulas: one must calculate the resolutions to understand the proof.

Another problem with resolution is its very uniformity. The fact of the matter is that while resolution will find a proof if one exists, this is not really a desirable feature. Instead, what is desired is a proof procedure that proves plausible inferences relatively quickly. Such a procedure need not be complete so long as it is implementable and does a reasonably good job in proving fairly routine theorems. The issue here is not mechanical mathematics, but rather the basis for a practical English language understanding system. The fact that a proof procedure is not complete does not matter so long

as the procedure produces proofs in a reasonable length of time for some class of formulas that includes most of the formulas that can reasonably be expected in practice. Unfortunately, very little is known about the bounds on the complexity of mechanical theorem-proving procedures: this is a research topic in the study of mechanized logic.

Yet another objection to resolution as a basis for a natural language question-answering system is that the resolution procedure is primarily syntactic in nature. The machine manipulates the symbols in the formulas in order to make the resolutions, but does not really use semantic information. Moreover, almost all of the editing strategies that are used by theorem-provers to make practical the process of finding a proof depend on syntactic properties of the formulas such as their length, the names of the variables that occur in them, and their syntactic form. Except for the work of Hayes and his associates there seems to have been relatively little effort to use semantic information to help reduce the number of clauses appearing at the nodes of the tree or to simplify their form. As Nilsson says in [18]:

> The reader probably noticed that all of the search strategies discussed in this chapter involved syntactic rather semantic rules (that is, search restrictions and orderings were based on the form of the clauses and possible deductions rather than on their meaning). Semantic guidance could be provided in a number of ways, but there have not yet been many attempts in this direction.

This is exactly what we wish to avoid in our work. Our approach to computational linguistics involves crucially the notation of finding the meaning conditions for sentences, and using those in some manner to

determine the answer to input questions. Thus, it is highly desirable that our computational methods use this semantic information. It also seems plausible based on human experience that the theorem-proving process itself might be improved by the use of such information. Exactly how such information should be used in the theorem-proving process is difficult to see, but for the present time at least, the resolution method of theorem-proving does not incorporate the use of such information, and thus, is not suitable for use as a basis for question-answering in a system that is attempting to understand natural language input.

While we have said a good bit about what information extraction is not, we have not said much about what it is and how one would implement it. The basic idea is that information extraction is to be a heuristic procedure that operates from a data base of mathematical and linguistic facts. As we noted above, information extraction has two distinct uses, and these two different uses to some extent determine what we actually mean by information extraction and how it is to be implemented. We shall begin, as is our custom, by considering the simpler case first- that of the use of information extraction to treat "how" questions.

IV.3 The Application of Information Extraction To How Questions

The treatment of "how" questions is based on the simple idea

that the mathematical functions that are used by our system to answer computational questions form a hierarchy. Some of these functions--such as addition, for example--are to be considered as mathematically primitive, and the other functions in the system are to be considered as complex. For instance, the FACTOR function is a relatively complex arithmetical function that is built up out of the division function. So to answer a question as to how one computes the factors of, say, 6, one would look at the FACTOR function as it is defined in our system, and notice that the LISP definition calls the DIVISION function. From the definition of the FACTOR function one could extract this information and use it to answer the "how" question. Note that in such a case the decision to use information extraction is automatic and is based on the type of question that one has to answer: "how" questions always call this type of semantic evaluation, and this fact should be incorporated into the toplevel linguistic functions that correspond to the English for "how".

There are two very important issues that remain regarding information extraction in this context. The first is the omnipresent problem of what primitives are to be chosen. Clearly, given the generality of the LISP interpreter it is possible to choose the same primitives that serve as the basic functions in the standard definition of the set of recursive functions, i.e., the zero function, the successor function, and the projection functions. However, this leads to rather messy function definitions. It seems somewhat more reasonable

for most applications to let the usual arithmetical operations be chosen as primitive and try to build the other arithmetical operations out of them. In fact, this is essentially what we have done in our implementation of arithmetic, for the AILISP interpreter is quite poor in pre-defined arithmetical functions, and most of the functions that we used other than the basic four operations we had to define ourselves. Of course just exactly the primitives that one chooses depend on the applications of the system that one has in mind.

The second problem that we must solve, and we have to admit that we have as yet not been successful in doing this, is the actual implementation of this idea. The fundamental idea involved in the implementation is to scan the definition of the function that we wish to know how to compute and look for the occurrence of one or more of the chosen primitive functions. If none of the primitives occur, then something is wrong, and an error routine is called. If only one primitive is found, then that is returned as the basis (at least) of an answer; if more than one primitive is found, the problem of how they are related is encountered, so that it is not so clear how to proceed. As a first approximation, one can simply return a list of all primitives found in the order in which they occur in the function definition. Clearly more powerful heuristics will be required later to interpret the LISP code and to provide better explanations than this. We shall give in our standard notation an outline of what our first proposal would look like.

```
EXTRACTPRIM (X) <= REVERSE (IF NULL (X) THEN NIL
ELSE IF MEMBER (CAR X, PRIMITIVES) THEN CONS (CAR X, EXTRACTPRIM CDR X)
```

ELSE EXTRACTPRIM CDR X)

where REVERSE is a LISP function that accepts a list as its argument and returns as its result that list with elements listed in reverse order.

IV.4 Information Extraction As an Aid to Problem Solving

Now we turn to a slightly more speculative venture--information extraction to assist in the actual computations themselves. As we noted before, the semantic evaluation program creates large numbers of functions as it executes. Since these functions are produced by functions that are hand coded, the resulting new functions have a rather standard format which lends itself to pattern-matching. The basic idea here is to do a better job computationally by using the mathematical relationships of the functions that are already defined in the system.

To make this point a little clearer, we shall give an example in which what we have in mind might make things a little easier for the computational program. Suppose that we are asked if the set of numbers greater than 5 and less than 3 contains a prime. In our present implementation about all that the program is able to do is to generate a characteristic function for the set: it does not know that the set of all numbers greater than 5 and less than 3 is empty. The semantics system, if modified to include this type of information extraction,

would scan the definition of the characteristic function for this set prior to applying it to arguments to check to see if the function might be simplified in some way. In this using the known mathematical relationship between greater than and less than, it would look for a pattern of the form

AND (GT X1) (LT X2).

If this pattern were found, then the values of X1 and X2 are compared and the set of all numbers between x1 and X2 (which may be empty) is generated. This gives us much more definite information about the set: in the particular case considered above the information is complete in the sense that it answers the question by itself. Even if there were some numbers in the generated set, we still have the information that the set is finite, and that the answer to many questions can be obtained by simply scanning the set. As we noted in Section II.3 it is almost always the case that a complete (or perhaps we should say a better formatted) answer is most easily obtained from the case in which we have an explicit list to work with rather than a characteristic function.

IV.5 Mathematical Information To Be Used By Information Extraction

There is of course the question of what kinds of mathematical information can and should be incorporated into the semantic evaluator

at this juncture. It seems likely that this can be answered only after some experience with the system and only after it has been determined what the system is to be used for. At the bare minimum it seems likely that one would want to incorporate information regarding the ordering relationships on the integers as well as information about the relation between a prime number and its factors. It also seems likely that information about non-explicit negations, e.g., prime and composite could be used by the information extraction routines.

The actual implementation of this depends first on the establishment of standard forms for the definition of functions in the semantics system; this has already been done. Then a pattern-matching capablity must be implemented as well as a method of using that capability in conjunction with a database of mathematical facts. This has not yet been attempted.

IV.6 The Heuristics Actually Used in the Semantic Evaluator

The next item that we shall consider is the problem of the heuristics that we actually used in the program that we have already written. For the most part, these are quite simple and rather obvious, but they greatly increase the fluency of the system and improve its computational ability. Most of these heuristics are based on rather elementary facts about number theory and set theory, and are programmed into the functions that used in a very direct manner. The usual method of calling the heuristic functions is to have the LISP function that is

going to use the heuristic call a special auxiliary function that decides if the heuristic is applicable. This function does the actual computation if the heuristic is applicable and returns to the toplevel function if not.

IV.7 Heuristics for the Set Theoretical Functions

Having given this general outline of the manner in which our heuristics are applied, we shall next go into rather great detail about the heuristics that we used and how they might be improved. The heuristics that are currently in the system can be divided into two groups. In the first group are heuristics that are associated with the set-theoretical functions that correspond to the English, and serve to simplify certain special cases of the set theory involved. To illustrate this, let us consider once again the I function. Here we have only two fairly simple heuristics. The first is that if the two arguments to the I function are equal according to LISP--which is a good bit stronger than set-theoretical equality--the I function returns the first argument. For clarity we shall give the definition of equality in LISP. By definition, the EQ function returns T if its two arguments are the same LISP atom, and NIL otherwise. Then we have:

```
EQUAL (X, Y) <= IF ATOM X OR ATOM Y THEN EQ (X, Y)
ELSE EQUAL (CAR X, CAR Y) AND EQUAL (CDR X, CDR Y).
```

We shall show that if two things are LISP equal, then they are set-theoretically equal.

Theorem: Let X and Y be s-expressions. If EQUAL (x, y) then X and Y are set-theoretically equal. The converse is false.

Proof: Suppose that EQUAL (X, Y). If X and Y are atoms, then even if X and Y represent characteristic functions, we have that X and Y are the same set or are characteristic functions for the same set. If X and Y are lists, then clearly X and Y have the same members, and hence are set-theoretically equal.

This should be sufficient to justify this heuristic. It is worth noting that using this heuristic does not require much computation time for all that is involved is a simple check for equality. The other heuristic that is used with the I function is simply a check for the empty set. This check is not very good, for it checks only for the empty set represented as an explicitly empty LST. Of course, this has the form (LST) rather than being simply NIL. Again, the check is quite inexpensive computationally. Similar heuristics are used with each of the basic set-theoretic functions of the system--U, S, and SD.

IV.8 Heuristics For Doing Arithmetic

There are other heuristics that are used to produce better answers to questions than can be obtained from simply manipulating the characteristic functions, and lists that are given. Most of these are based on simple facts of arithmetic. Although at present there are heuristics only for such things as how the set of odd numbers and the

set of even numbers are related, it is clearly an important task to find and to incorporate other such arithmetical facts. From experience with the program, it is clear that the use of even very limited heuristics greatly increases the fluency and the apparent performance of the program. All of these heuristics are implemented by storing the desired result of some computation on the property list of one of the arguments to the computation and then having the function look up that result. Again, we shall consider the I function. Since these heuristics are used only when we have two sets represented by their characteristic functions, the heuristics are called only after it is determined that the I function has received two characteristic functions as arguments. The sets to which the heuristics are applicable are marked by having on the property list of the atom that names their characteristic function under the property ISSPECSTS the value T. A simple table lookup routine is called to check for applicability of the heuristic. If the applicability function--known as ISSPCSTS--returns T, then the answer is found by looking on the property list of the first argument to the I function under a property named the second argument. We shall give the details of the definition in the usual manner, and shall as is our custom omit all of the irrelevant parts of the definition. The function that we shall define is called ISTS, and is called from the I function: this function computes the intersection of sets when the sets are represented as characteristic functions. (This function is also defined and is more completely discussed in Section III.10.)

```
ISTS (X, Y) <= IF ISPCSTS (X, Y) THEN SPCSTS (I, X, Y) ELSE
DEFINEFUNCTION (W, MAKEINTERSECTION (X, Y))
```

The function SPCSTS is defined to be the value of the property named by the second argument on the property list of the first. Note that this implies that each fact must be stored twice for intersection is commutative. For example, the fact the intersection of the odd numbers and the even numbers is empty must be stored both under the property of odd on the property list of even and also on the property list of odd under the property even. Since there are also the U, S, and SD operations to consider, there must be some mechanism for storing the heuristics that are needed for all of these on the same property list under the same property. This is done by marking each of the items in the list under a particular property by the operation with which it belongs. Then each operation merely indexes into the list by means of an association function to retrieve the correct answer. Assume that the GET function is a LISP function of 2 arguments. The first of these is the name of the identifier whose property list we wish to access while the second of these is the name of the property whose value we wish to retrieve. Then if ASSOC is a LISP function that retrieves from a list the first element whose CAR is equal to the given element, we have the following definition for the function SPCSTS.

```
SPCSTS (X1, X2, X3) <= ASSOC (X1, GET (X2, X3))
```

This particular design of the data structure has the advantage of allowing uniformity over all of the operations. Thus, we only need

this function to implement all of the heuristics that are involved with the relationship of sets of numbers to each other. The database upon which this function draws must of course be set up in advance of running the program: this is done by the use of a rather simple SAIL program that generates the proper s-expressions which are then read into the LISP core image when the semantics system is created.

IV.9 The Justification of the Heuristics

The most important issue, however, regarding the heuristics is the problem of how to justify them. From a mathematical point of view this is completely trivial, for these facts among the most elementary results of number theory. Specifically, we have incorporated the results of the set-theoretical union, intersection, set-difference, and subset operations on the sets EVEN, ODD, PRIME, and NUMBER. In some cases this involves the addition of special pre-coded characteristic functions such as ODDPRIME to the system. In other cases such as the intersection of the EVEN numbers and the PRIMEs this means storing the structure (LST 2). However, the use of these heuristics must be justified computationally on the basis of the performance of the system, and also on the basis of some kind of mathematical naturalness. This last constraint is necessary if the system is to be able to answer "how" questions in a reasonable way.

With regard to the performance of the system these few heuristics greatly improve its performance. The major problem that the

semantic evaluator has when given a reasonable question to answer is how to represent the sets in the semantic parse. The use of these heuristics enable the system in a number of cases to replace the use of characteristic functions by explicit lists. Since as was pointed out in Section II.3, it is almost always desirable to use explicit lists rather than the characteristic functions, and since this tends to make the system look more fluent, the apparent performance of the system is enhanced by the use of these heuristics. For example, the representation of the set of even primes by the explicit list (LST 2) is more perspicuous and makes it easier to do further manipulations on the set. It is important to note that while the representation makes no difference in the mathematics involved, it is very important in the actual computer implementation. The characteristic functions are, in some sense, less desirable as a representation than the explicit lists simply because many of the manipulations of the characteristic functions that we perform are simply the storing of the current operation until such time as we have an explicit list representation and are actually able to carry out the operation. For example, when we take the intersection of two sets represented as characteristic functions all that happens is that a new characteristic function is created which can be used to check for membership in the newly created set. When we take the intersection of two sets that are represented as explicit lists, however, then we actually get a list as the result, and it is such a list that we are always striving to get if possible in any

of our computations. Note that this problem of representation is not handled by classical mathematical tools thus indicating a need for some new mathematics to formalize this problem.

Chapter V

Comparisons With Other Work

V.1 Machine Translation

There have in the past twenty years been many attempts to program the computer to understand natural language input. Indeed, this was one of the first problems attacked in the then young field of artificial intelligence during the 1950's. At that time the emphasis was on machine translation--attempting to get the machine to translate documents from another language (usually Russian) into English. After ten years of work on this subject interest in it died down in the mid-1960's when it was concluded that the project of machine translation seemed to be hopeless. Although a number of programs had been written that could translate some fixed piece of text (usually a demonstration document upon which the program had been debugged), when given any other material as input, these programs failed to produce any translation at all. Usually, the theory behind such programs was syntactic: the syntax was analyzed according to some scheme or other and then a word for word match was made on the syntactic components.

Towards the end of the machine translation era, transformational grammar became popular, leading to work like that of Petrick [9] and Kuno [19]. These workers attempted to write parsers

for natural language based on the syntactic theories of Chomsky and others. Although these programs could often many parses for a given input sentence, they did not have any semantic component, and their only function was to parse input. Moreover, most of the parsing algorithms that had to be used were quite inefficient (see Woods [10] on this point), and have been replaced in more recent systems by things like augmented transition network parsers.

V.2 The History of Question-Answering Systems

At about the same time as this work on transformational syntax was being done, mostly at Harvard and Mitre, computer scientists became interested in question-answering systems, and in particular, in question-answerers that accepted natural language questions. The early work in this field is described by Simmons in [20], and much of the original work has been collected in [21] by Minsky. Clearly, such systems needed to be much more concerned about the semantics of the input question than about the syntactic analysis of English, and in that respect they resemble our work. However, these programs were for the most part oriented toward problem-solving using the computer rather than toward the understanding of English. Although programs such as that of Bobrow [21] accepted a modified form of English, they generally looked for certain words to be matched against patterns of pre-stored keywords. From these patterns the computer could in the case of Bobrow's program set up simultaneous equations to solve. Clearly,

there was no general theory of language and of meaning that was used to design these programs.

More recently, there have been a number of attempts to develop a theory of English and to use that as a basis for question answering systems: a good survey is to be found in the more recent article by Simmons [22]. Of these attempts we shall consider the work of only four in any detail.

V.3 The Work of Winograd on the BLOCKS Program

Probably the best known of the natural language processing programs that is currently under active development is that of Winograd [23] and [24]. Since the entire philosophy behind the BLOCKS program is quite different from that which guided our own work, it is necessary to explain a little bit of the world view that is prevalent at the MIT AI Lab where Winograd wrote his program. The MIT AI researchers believe that the only adequate theory of a part of artificial intelligence is a program, and that, furthermore, a program that achieves a certain level of apparent intellectual competence is to be regarded as a theory of intelligence in that area. Thus, Winograd's program is to be regarded as a theory of natural language processing, and any description of it other than the actual code is simply a description of a theory of natural language. Thus, the theory is heavily dependent on the implementation. This is exactly the opposite of our approach on this matter: we have explicitly attempted to

develop a machine independent theory of natural language processing and understanding. Although the major topic of the present dissertation has been the implementation, the important point about the implementation is that it is a source of insight about natural language and leads to the development and to the testing of theories of language. Indeed, the most unsatisfactory part of our work from this point of view is not the incomplete state of the implementation, but the lack of a fully developed theory of what the program does and should do. Despite our attempts to deal with this in previous chapters, much remains to be done with most of the work actually a part of the development of the mathematical theory of computation.

The theory of grammar that is used by Winograd is systemic grammar which was developed by Halliday and modified by Winograd. We shall, however, base our discussion solely on the presentation of it that is made by Winograd in [24]. Systemic grammar is a rejection of the basic approach of grammarians since Chomsky in favor of a system of nodes representing the basic elements of the sentence and having associated with them features of various kinds. Each node has a type and a list of features that are either present or absent in this particular node. This scheme leads to rather flat parse trees each of whose nodes have relatively large amounts of information associated with them. The major advantages of this scheme is that it is easy to write a parsing program using it and the parses that are produced are very informative in that there is a great deal of syntactic structure

that is discovered and stored during the parsing process. Since Winograd does not give his exact grammar in his published work and since any details of it are really not to the point here, we shall simply summarize our understanding of it. It seems to be a rather intuitively simple rendering of most of the basic syntactic features of that part of English with which he dealt.

The parsing scheme that is used in the BLOCKS program is of some interest to us, for it interacts with the semantic portion of the program in many ways. The exact nature of this interaction will be discussed presently, but first we must say something about PROGRAMMAR, the parsing scheme that is used by Winograd. PROGRAMMAR is a derivative of the programming language LISP, and is used by Winograd to program his grammar. The elements of the grammar are embodied in procedure definitions. The parse is essentially bottom-up with some variations in the flow of control in the form of interrupts. In particular the semantics routines and special heuristics may be called by the parser when an appropriate structure is encountered by the parser. It would seem to be possible to describe Winograd's parsing scheme by Floyd-Evans production language using an additional pushdown list as a context stack, but we have not worked out the details of this.

The semantic component of Winograd's program uses the capabilities of the very powerful programming language PLANNER, which was developed at MIT by Carl Hewitt. PLANNER is a pattern matching and

backtracking language that is also embedded within LISP. Wingorad has analyzed the semantic domain that he is dealing with--the world of toy blocks--and coded his results as a special type of PLANNER procedure, called a theorem. The program performs logical inferences by pattern-matching using the pattern-matching language MATCHLESS. Winograd's semantic program maintains a history of the discourse, an internal map representing the locations of blocks in space and a set of PLANNER theorems describing these and relating English language commands to this database.

V.4 The Predicate Calculus as Deep Structure--The Work of Sandewall

It should be clear that Winograd's program is designed to do something very different from ours in that it maintains and manipulates a data base rather than using computation to solve problems that are posed in natural language. The next piece of work that we shall consider is that of Sandewall which is quite different from either the work of Winograd or our own. Sandewall is seeking a machine independent theory of natural language, but one whose primary concern is natural language processing by computer rather than psychological or philosophical explication of some abstract process of natural language understanding. Once this is understood, it is easy to see why Sandewall's ideas are so attractive. According to him, there is a deep structure for English which has a very definite form--first order

predicate calculus. Apparently, Sandewall believes that the syntactic surface structure of English can be translated into a first-order formalism or something slightly more powerful like a multi-sorted predicate calculus. Once this is done, resolution theorem-proving such as is common at present can then be applied to answer any input questions and to determine the consistency or inconsistency of input declaratives. There are, of course, several problems with this line of attack. The first is the current state of theorem-proving technology: at present resolution theorem provers are too slow and too inefficient to be used for a practical question-answering system; for further comments on this see Section IV.2. Although Sandewall expresses some hope that the resolution method can be made more efficient, this remains to be seen. Another much more severe problem with Sandewall's approach is that he seems to provide no general method for translating from the syntactic surface structure of an input to his putative deep structure, and indeed, it is not very clear what the relationship should be. It is clear, however, as we pointed out before, that the heuristics that are used in elementary symbolic logic courses are too simple to do general translation from the surface structure to such a deep structure. For one thing it is difficult to spell out precisely exactly what the heuristics that are actually used are.

V.5 The Psychological Model Approach of Schank

While the intentions of Sandewall are primarily motivated by

computer science considerations, the work of Schank is based primarily on ideas about psychology and linguistics. However, Schank regards a computer implementation of a theory to be the best available test of it. (Schank's work has been reported and discussed in a number of places, including a large number of Stanford Artificial Intelligence Memos. See, for example, [25] and [26].)

In many ways the work of Schank is similar to our own. In particular, he is committed to a machine independent theory of natural language processing rather than a program as a theory. Moreover, he regards semantics as being the key to natural language understanding. However, Schank's views on semantics are quite different from our own, and he does not really believe in using syntax although his programs do use it as a matter of convenience. According to Schank, there are psychological structures that underlie all of human language. These are not linguistic structures, but serve as a base onto which linguistic structures can be mapped: these structures Schank calls conceptual dependency relations. For the exact format of these see [25] or [26]. The basic idea is that the meaning of natural language sentences can be represented by abstracting their features into a relatively small number of primitives. Each sentence is assumed to describe an act, which has an agent performing one of twelve or fourteen primitive acts: for Schank the only primitives are acts which are generally verbs in more standard treatments. In addition, a sentence may indicate causal relations and instrumentalities. What is

important about this is less the exact structure than the manner in which Schank seeks to use his representations. The crucial notion is that of inference.

Logicians have given the notion of inference a very precise meaning in the last hundred years, i.e., one sentence can be inferred from another just in case in every model for the second sentence the first also comes out to be true. This is very definitely not what Schank means by inference. Instead, one sentence can be inferred from another if whenever a person knows that the second holds, he would generally also believe in the first. Thus, inference is to be based on the common knowledge of language and the world that it describes as represented in terms of conceptual dependency. It is very important to realize that one major goal here is to have any program that uses conceptual dependency make the same types of mistakes that a human being would make. In contrast to this our notion of inference is at present at least based upon the classical notions used in logic. It should be pointed out the purpose of our system is to answer questions about elementary mathematics while Schank's is designed to be used to carry on an ordinary conversation.

Not only does Schank's system have a notion of inference in it but it also involves a model of memory. An attempt is being made to handle the understanding of language in context rather than on a sentence-by-sentence basis. While it is not yet possible to give a detailed description of this memory model, the basic idea is that the

program is to keep on a short term basis the details of input and is to encode and to store into a long term memory the important parts of the conversation. Clearly this encoding process is going to have to be rather clever in order to retain sufficient information to maintain the thread of the discourse without taking too much memory space. In our work we have not yet attempted to provide the system with memory although we do envision an implementation of declaratives that will involve the maintenance of a data base of facts. Input declaratives will be compared with the current memory for consistency, and some method of deciding what to store and what to eliminate will have to be worked out.

The major formal difference between our work and that of Schank should then be clear: there is no precise relationship between syntax and semantics in his system while in ours the syntax is used as a framework for laying out the semantics of the surface structure (as opposed to the deep structure which is obtained by transformations). It is also true that the types of representations that we use for our semantic structures are very different from Schank's conceptual dependency diagrams. In our current development the program itself uses LISP s-expressions to represent the semantics, but we hope to develop a system of function schemata that can be used to represent the semantic parses. It is also true that conceptual dependency removes more of the surface structure than our semantic parses do. For example, in a sentence such as

John hurt Mary (a)

the conceptual dependency diagram is roughly "John did something to put Mary into the hurting state." The reason for this is that there is no primitive hurting act. In our system the representation would be something like (HURT JOHN MARY) where HURT is assumed to be a predefined function.

V.6 The Work of Woods on Natural Language Processing

The work of Woods in natural language processing is closer to ours than to Schank's. Woods, whose work has been reported in [10] and [27], has developed a very successful parsing scheme for sentences of natural language called the augmented transition network. In addition, he has written semantic systems which answer questions about airline reservations and about lunar geology.

As this thesis is not concerned primarily with parsing techniques and as it is very difficult to improve upon Woods' own exposition in [10] we shall not discuss his parsing scheme in very great detail. The basic idea is to extend transition diagrams from finite automata to devices that are able to parse an arbitrary recursively enumerable language. This is done by using a pushdown mechanism that allows a call to be made to other networks to handle the parsing of certain grammatical categories. This is enough to handle context-free languages, but to deal with more complex languages Woods adds predicates that must be satisfied before a transition can be made.

The resulting parsing scheme is shown to be as efficient as a

parsing scheme for context-free languages. The only advantage that we can claim over Woods in this area is perspicuity and extendability. In our system the grammar is written in isolation from the implementation and is read in when the program is started. Woods ist, on the other hand, write new transition networks whenever he wishes to change the grammar.

The work of Woods on semantics seems to be somewhat similar to our own, but it has been published only recently, and is not described in sufficient detail to allow us to analyze it at this time. But it does seem clear that the field of computational linguistics is still an open one and that the definitive piece of work that will either make the computer fluent in natural language or show that such is impossible remains to be done.

Index

appositions 69
arithmetical relations 70

ASSOC 99

asynchronous processing 63

ATOM 53

augmented transition network 104

backtracking 61, 108
basic functions 91

BLOCKS 105

calling sequence 58, 63

CAR 52
CDR 52

characteristic function 68
characteristic functions 56, 97

CHL 67, 68
CHOICE 62
Chomsky 11
Chomsky normal form 7

computational ability 95

CONS 52
CONSTRUCT 5, 64

constructive sets 18
context- sensitive 13
context-free grammar 11
context-free language 26
control structures 23, 33

data type 60
declaratives 66
deep structure 108
demons 63

DIV 70

division 91

elementary mathematical
language 2

EQ 71
EQL 71
EQL1 71
EQL2 71
EQUAL 71

equality 36, 96
equivalence of schemata 44
equivalent 62
escalation of type 20

EVAL 59, 65, 68

evaluation 12, 35

EVEN 100
EXIST 81

explicit list 15

EXPR 54
EXTHNP 81

FACTOR 91
FAILURE 62
FCN 68
FCNMK 83
FEXPR 54, 71

first order logic 28

FOR 53
FRACTION 70

function definitions 56

GE 72

generative grammar 11

GET 65, 99
GT 72

habitability problem 8
heuristic procedure 90
heuristics 76, 85, 95

I 61, 72, 81, 96, 98
ICHL 73
ILST 74

information extraction 85

interrupts 63, 107
intersection 100

ISTS 74, 98

keywords 104

L70 8
LE 72

linguistics 110

LISP 7, 50, 65

list 52

LIST 53
LST 60
LT 72

machine independent theory 108
machine translation 103

MATCHLESS 108

mathematical functions 22, 32, 54
mathematical information 94

MLISP 50
MLISP2 8

model structure 13

natural deduction calculus 88
natural language input 1
natural language output 1, 6

NIL 52

non-explicit negations 95
non-transformational
schemata 35, 40

Nondeterministic algorithms 62
NULL 53
NUMBER 100
NUMBERP 53, 70

ODD 100
ODDPRIME 100

ordering relationships 95
output 66

parsing 26
pattern matching 107
pattern-matching 85, 93

PLANNER 86, 107

potentially denoting grammar 13
predicate calculus 109

PRIME 100
PROGRAMMAR 107

property list 98
propositional logic 88

psychology 110

question-answering system 2
question-answering systems 104

QUOTE 65

registers 35
resolution 87, 109

S 76, 78, 82, 97

s-expression 52

SAIL 7, 65

schemata 33, 44

SD 76, 79, 97

semantic deep
structure 23, 27, 32
semantic transformations 27
sentence typing functions 66
set theory 17
set-difference 100
side-effect 61
side-effects 55

SPCSTS 99

speech recognition 7

STS 68

subset 100

SUCCESS 62

surface structure 54, 109
syntactic transformations 27
systemic grammar 106

TENEX 6

theory of language 3
transformation 25, 60
transformational grammar 103
transformational schemata 34, 40
transformations 72
translation 29

TV 68

U 76, 77, 97
ULST 77
ULSTSTS 77

union 100

UNT 68
UNVHNP 82
USTS 77

valuation function 12
variables 35

Winograd 63

References

1. Irons, E. T., Towards more versatile mechanical translators, Proceedings of Symposia on Applied Mathematics, vol. 15 ,pp.41-50, American Mathematical Society, Providence, Rhode Island, 1963.

2. Knuth, D. E., Semantics of context-free languages, Mathematical Systems Theory 2 (1967), pp. 127-145.

3. Coles, L. S., Techniques for information retrieval using an inferential question-answering with natural language input, Artificial Intelligence Center Technical Note 74, Stanford Research Institute, 1972.

4. Hewitt, Carl, The description and theoretical analysis of PLANNER, Doctoral dissertation, MIT, 1971. Also AI TR-258, MIT Artificial Intelligence Laboratory, 1972.

5. Hopcroft, John, and Ullman, J. D., Formal Languages Their Relation to Automata, Addison-Wesley, Reading, Massachusetts, 1969.

6. Smith, Robert L., The syntax and semantics of Erica, Doctoral dissertation, Stanford University, 1972. Also Technical Report 185, Institute for Mathematical Studies in the Social Sciences, Stanford University, 1972.

7. Suppes, Patrick, Semantics of context-free languages, Technical Report 171, Institute for Mathematical Studies in the Social Sciences, Stanford University, 1971.

8. Smith, Nancy W., A grammar for a fragment of elementary mathematical language, Doctoral dissertation, Stanford University, 1973.

9. Petrick, S., A recognition procedure for transformational grammars, Doctoral dissertation, MIT, 1965.

10. Woods, W. A., Transition network grammars for natural language analysis, Communications of the Association for Computing Machinery 13 (1970), pp. 591-606.

11. Early, J. W., An efficient context-free parsing algorithm, Communications of the Association for Computing Machinery 13 (1970), pp. 94-102.

12. Sandewall, E. J., Formal methods in the design of question answering systems, Artificial Intelligence, 2 (1971), pp. 129-145.

13. Paterson, M. S., and Hewitt, C., Comparative schematology, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Association for Computing Machinery, New York, 1970.

14. Smith, David Canfield, MLISP, Artificial Intelligence Memo 135, Stanford Artificial Intelligence Laboratory, 1970.

15. Quam, Lynn H., and Diffie, Whitfield, Stanford LISP 1.6 Manual, Artificial Intelligence Operating Note 28.7, Stanford Artificial Intelligence Laboratory, 1971.

16. Smith, David Canfield, and Enea, Horace J., MLISP2, Artificial Intelligence Memo 195, Stanford Artificial Intelligence Laboratory, 1973.

17. Floyd, R. W., Nondeterministic algorithms, Journal of the Association for Computing Machinery 14 (1967), pp.636-644.

18. Nilsson, Nils J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.

19. Kuno, S., A system for transformational analysis, Report NSF-15, Computation Laboratory, Harvard University, Cambridge, Massachusetts, 1965.

20. Simmons, Robert, Answering English questions by computer: a

survey, Communications of the Association for Computing Machinery, 8 (1965), PP. 53-70.

21. Minsky, Marvin, editor, Semantic Information Processing, MIT Press, Cambridge, Massachusetts, 1968.

22. Simmons, Robert, Natural language question-answering systems: 1969, Communications of the Association for Computing Machinery 13 (1970), pp. 15-30.

23. Winograd, T., Procedures as a representation for data in a computer program for understanding natural language, Doctoral dissertation, MIT, 1970.

24. Winograd, T., Understanding Natural Language, Academic Press, New York, 1972.

25. Schank, Roger, Conceptual dependency: a theory of natural language understanding, Cognitive Psychology 3 (1972), pp. 552-631.

26. Schank, Roger, The Fourteen Primitive Acts and Their Inferences, Artificial Intelligence Memo 196, Stanford Artificial Intelligence Laboratory, 1973.

27. Woods, William A., Semantics for a question-answering system, Doctoral dissertation, Harvard University, 1967.

FILMED FROM BEST AVAILABLE COPY

(Continued from inside front cover)

165 L. J. Hubert. A formal model for the perceptual processing of geometric configurations. February 19, 1971. (A statistical method for investigating the perceptual confusions among geometric configurations. Journal of Mathematical Psychology, 1972, 9, 389-403.)

166 J. F. Juola, I. S. Fischler, C. T. Wood, and R. C. Atkinson. Recognition time for information stored in long-term memory. (Perception and Psychophysics, 1971, 10, 8-14.)

167 R. L. Klatzky and R. C. Atkinson. Specialization of the cerebral hemispheres in scanning for information in short-term memory. (Perception and Psychophysics, 1971, 10, 335-338.)

168 J. D. Fletcher and R. C. Atkinson. An evaluation of the Stanford CAI program in initial reading (grades K through 3). March 12, 1971. (Evaluation of the Stanford CAI program in initial reading. Journal of Educational Psychology, 1972, 63, 597-602.)

169 J. F. Juola and R. C. Atkinson. Memory scanning for words versus categories. (Journal of Verbal Learning and Verbal Behavior, 1971, 10, 522-527.)

170 I. S. Fischler and J. F. Juola. Effects of repeated tests on recognition time for information in long-term memory. (Journal of Experimental Psychology, 1971, 91, 54-58.)

171 P. Suppes. Semantics of context-free fragments of natural languages. March 30, 1971. (In K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes (Eds.), Approaches to natural language. Dordrecht: Reidel, 1973. Pp. 221-242.)

172 J. Friend. INSTRUCT coders' manual. May 1, 1971.

173 R. C. Atkinson and R. M. Shiffrin. The control processes of short-term memory. April 19, 1971. (The control of short-term memory. Scientific American, 1971, 224, 82-90.)

174 P. Suppes. Computer-assisted instruction at Stanford. May 19, 1971. (In Man and computer. Proceedings of international conference, Bordeaux, 1970. Basel: Karger, 1972. Pp. 298-330.)

175 D. Jamison, J. D. Fletcher, P. Suppes, and R. C. Atkinson. Cost and performance of computer-assisted instruction for education of disadvantaged children. July, 1971.

176 J. Offir. Some mathematical models of individual differences in learning and performance. June 28, 1971. (Stochastic learning models with distribution of parameters. Journal of Mathematical Psychology, 1972, 9(4),)

177 R. C. Atkinson and J. F. Juola. Factors influencing speed and accuracy of word recognition. August 12, 1971. (In S. Kornblum (Ed.), Attention and performance IV. New York: Academic Press, 1973.)

178 P. Suppes, A. Goldberg, G. Kanz, B. Searle, and C. Stauffer. Teacher's handbook for CAI courses. September 1, 1971.

179 A. Goldberg. A generalized instructional system for elementary mathematical logic. October 15, 1971.

180 M. Jerman. Instruction in problem solving and an analysis of structural variables that contribute to problem-solving difficulty. November 12, 1971. (Individualized instruction in problem solving in elementary mathematics. Journal for Research in Mathematics Education, 1973, 4, 6-19.)

181 P. Suppes. On the grammar and model-theoretic semantics of children's noun phrases. November 29, 1971.

182 G. Kreisel. Five notes on the application of proof theory to computer science. December 10, 1971.

183 J. M. Moloney. An investigation of college student performance on a logic curriculum in a computer-assisted instruction setting. January 28, 1972.

184 J. E. Friend, J. D. Fletcher, and R. C. Atkinson. Student performance in computer-assisted instruction in programming. May 10, 1972.

185 R. L. Smith, Jr. The syntax and semantics of ERICA. June 14, 1972.

186 A. Goldberg and P. Suppes. A computer-assisted instruction program for exercises on finding axioms. June 23, 1972. (Educational Studies in Mathematics, 1972, 4, 429-449.)

187 R. C. Atkinson. Ingredients for a theory of instruction. June 26, 1972. (American Psychologist, 1972, 27, 921-931.)

188 J. D. Bonvillian and V. R. Charrow. Psycholinguistic implications of deafness: A review. July 14, 1972.

189 P. Arabie and S. A. Boorman. Multidimensional scaling of measures of distance between partitions. July 26, 1972. (Journal of Mathematical Psychology, 1973, 10,)

190 J. Ball and D. Jamison. Computer-assisted instruction for dispersed populations: System cost models. September 15, 1972. (Instructional Science, 1973, 1, 469-501.)

191 W. R. Sanders and J. R. Ball. Logic documentation standard for the Institute for Mathematical Studies in the Social Sciences. October 4, 1972.

192 M. T. Kane. Variability in the proof behavior of college students in a CAI course in logic as a function of problem characteristics. October 6, 1972.

193 P. Suppes. Facts and fantasies of education. October 18, 1972. (In M. C. Wittrock (Ed.), Changing education: Alternatives from educational research. Englewood Cliffs, N. J.: Prentice-Hall, 1973. Pp. 6-45.)

194 R. C. Atkinson and J. F. Juola. Search and decision processes in recognition memory. October 27, 1972.

195 P. Suppes, R. Smith, and M. Leveillé. The French syntax and semantics of PHILIPPE, part 1: Noun phrases. November 3, 1972.

196 D. Jamison, P. Suppes, and S. Wells. The effectiveness of alternative instructional methods: A survey. November, 1972.

197 P. Suppes. A survey of cognition in handicapped children. December 29, 1972.

198 B. Searle, P. Lorton, Jr., A. Goldberg, P. Suppes, N. Ledet, and C. Jones. Computer-assisted instruction program: Tennessee State University. February 14, 1973.

199 D. R. Levine. Computer-based analytic grading for German grammar instruction. March 16, 1973.

200 P. Suppes, J. D. Fletcher, M. Zanotti, P. V. Lorton, Jr., and B. W. Searle. Evaluation of computer-assisted instruction in elementary mathematics for hearing-impaired students. March 17, 1973.

201 G. A. Huff. Geometry and formal linguistics. April 27, 1973.

202 C. Jensema. Useful techniques for applying latent trait mental-test theory. May 9, 1973.

203 A. Goldberg. Computer-assisted instruction: The application of theorem-proving to adaptive response analysis. May 25, 1973.

204 R. C. Atkinson, D. J. Herrmann, and K. T. Wescourt. Search processes in recognition memory. June 8, 1973.

205 J. Van Campen. A computer-based introduction to the morphology of Old Church Slavonic. June 18, 1973.

206 R. B. Kimball. Self-optimizing computer-assisted tutoring: Theory and practice. June 25, 1973.

207 R. C. Atkinson, J. D. Fletcher, E. J. Lindsay, J. O. Campbell, and A. Barr. Computer-assisted instruction in initial reading. July 9, 1973.

208 V. R. Charrow and J. D. Fletcher. English as the second language of deaf students. July 20, 1973.

209 J. A. Paulson. An evaluation of instructional strategies in a simple learning situation. July 30, 1973.

210 N. Martin. Convergence properties of a class of probabilistic adaptive schemes called sequential reproductive plans. July 31, 1973.

FILMED FROM BEST AVAILABLE COPY

(Continued from inside back cover)

211 J. Friend. Computer-assisted instruction in programming: A curriculum description. July 31, 1973.

212 S. A. Weyer. Fingerspelling by computer. August 17, 1973.

213 B. W. Searle, P. Lorton, Jr., and P. Suppes. Structural variables affecting CAI performance on arithmetic word problems of disadvantaged and deaf students. September 4, 1973.